目　　录

第一篇　常用数字信号的产生

第二篇　数字信号处理

第三篇　随机数字信号处理

第四篇　数字图像处理

第一篇　常用数字信号的产生

第一章　数字信号的产生

§1.1　均匀分布的随机数

一、功　能

产生（a，b）区间上均匀分布的随机数。

二、方法简介

均匀分布的概率密度函数为

$$f(x)=\begin{cases}\dfrac{1}{b-a} & ,a\leqslant x\leqslant b\\ 0 & ,\text{其它}\end{cases}$$

通常用U (a,b) 表示。均匀分布的均值为 $\dfrac{a+b}{2}$，方差为 $\dfrac{(a-b)^2}{12}$。

产生均匀分布随机数的方法如下：

首先，由给定的初值 x_0，用混合同余法

$$\begin{cases}x_i=(ax_{i-1}+c)\ (\mathrm{mod}\ M)\\ y_i=x_i/M\end{cases}$$

产生（0，1）区间上的随机数 y_i。其中：$a=2045$，$c=1$，$M=2^{20}$；然后，通过变换 $z_i=a+(b-a)y_i$ 产生 (a,b) 区间上的随机数 z_i。

三、使用说明

1. 子函数语句

double uniform（a，b，seed）

2. 形参说明

a ——双精度实型变量。给定区间的下限。

b ——双精度实型变量。给定区间的上限。

seed——长整型指针变量。*seed 为随机数的种子。

四、子函数程序（文件名：uniform.c）

```
double uniform(a,b,seed)
```

```
double a,b;
long int  *seed;
{ double t;
    *seed=2045*(*seed)+1;
    *seed=*seed-(*seed/1048576)*1048576;
   t=(*seed)/1048576.0;
   t=a+(b-a)*t;
   return(t);
}
```

五、例　题

产生 50 个 0 到 1 之间均匀分布的随机数。

主函数程序(文件名:uniform.m):

```
#include "stdio.h"
#include "uniform.c"
main()
{ double a,b,x; int i,j;
  long int s;
  double uniform(double, double, long int *);
  a=0.0; b=1.0; s=13579;
  for (i=0;i<10;i-+)
     { for(j=0;j<5;j++)
          { x=uniform(a,b,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
 }
```

运行结果:

```
0.4826355    0.9895945    0.7206707    0.7715826    0.8864250
0.7391634    0.5891514    0.8145781    0.8121262    0.7979975
0.9048367    0.3909597    0.5126686    0.4073076    0.9440937
0.6716261    0.4753571    0.1051798    0.0926647    0.4993505
0.1718712    0.4765749    0.5956669    0.1387815    0.8082657
0.9033289    0.3075924    0.0264425    0.0749702    0.3141527
0.4423084    0.5207319    0.8967896    0.9346323    0.3230572
0.6519232    0.1829033    0.0372286    0.1324558    0.8721647
0.5768661    0.6912775    0.6624966    0.8054800    0.2066078
```

0.5129900 0.0645466 0.9977674 0.4344330 0.4154520

§1.2 正态分布的随机数

一、功 能

产生正态分布 $N(\mu,\sigma^2)$ 的随机数。

二、方法简介

正态分布的概率密度函数为

$$f(x)=\frac{1}{\sqrt{2\pi}\sigma}e^{-(x-\mu)^2/2\sigma^2}$$

通常用 $N(\mu,\sigma^2)$ 表示。式中 μ 是均值，σ^2 是方差。正态分布也称为高斯分布。

产生正态分布随机数的方法如下：

设 r_1，r_2，…，r_n 为（0，1）上 n 个相互独立的均匀分布的随机数，由于 $E(r_i)=\frac{1}{2}$，$D(r_i)=\frac{1}{12}$，根据中心极限定理可知，当 n 充分大时

$$x=\sqrt{\frac{12}{n}}\left(\sum_{i=1}^{n}r_i-\frac{n}{2}\right)$$

的分布近似于正态分布 $N(0,1)$。通常取 $n=12$，此时有

$$x=\sum_{i=1}^{12}r_i-6$$

最后，再通过变换 $y=\mu+\sigma x$，便可得到均值为 μ、方差为 σ^2 的正态分布随机数 y。

三、使用说明

1. 子函数语句

```
double gauss(mean,sigma,seed)
```

2. 形参说明

mean ——双精度实型变量。正态分布的均值 μ。
sigma——双精度实型变量。正态分布的均方差 σ。
seed ——长整型指针变量。*seed 为随机数的种子。

四、子函数程序(文件名:gauss.c)

```
#include "uniform.c"
double gauss(mean,sigma,s)
double mean,sigma;
```

```
long int  *s;
{ int i; double x,y;
  double uniform();
  for (x=0,i=0;i<12;i++)
    x+=uniform(0.0,1.0,s);
  x=x-6.0;
  y=mean+x * sigma;
  return(y);
}
```

五、例　题

产生 50 个均值为 0、方差为 1 的正态分布的随机数。

主函数程序(文件名:gauss.m):

```
#include "stdio.h"
#include "gauss.c"
main()
{ int i,j; long int s;
  double x, mean, sigma;
  double gauss(double, double, long int * );
  mean=0.0; sigma=1.0; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=gauss(mean,sigma,&s);
          printf(" %13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
   2.8997211   -0.9088573    0.2041950   -0.2572155   -0.8516827
   0.7996998   -0.9866619    0.0431385   -1.9194927    0.2543507
  -0.3689251    1.2145863   -1.0537090    1.7050953   -1.6925945
  -0.4928722    1.9956684   -0.5980663    1.2923298    0.1707630
  -0.5213604   -0.4051342    0.8358479   -0.5445080    1.6452045
   0.5338917   -0.8120403   -0.3886852   -0.2546368    0.4690113
  -0.4013348   -0.1117687   -0.9708843    0.6502247    1.3179646
   0.5362415    0.7464619    1.3275318   -0.4041424    1.8053455
  -0.8525982   -0.2490673    1.6823444    0.9455433    0.4819355
```

1.1704273　　−0.1725750　　0.2068348　　−1.9999371　　0.8360157

§1.3　指数分布的随机数

一、功　能

产生指数分布的随机数。

二、方法简介

1. 产生随机变量的逆变换法

定理　设 $F(x)$ 是任一连续的分布函数，如果 $u\sim U(0,1)$ 且 $\eta=F^{-1}(u)$，那么 $\eta\sim F(x)$。

证明　由于 $u\sim U(0,1)$，则有

$$P(\eta\leqslant x)=P(F^{-1}(u)\leqslant x)=P(u\leqslant F(x))=F(x)$$

所以，$\eta\sim F(x)$。定理证毕。

此定理给出了从均匀分布随机数到给定分布 $F(x)$ 的随机数的变换。根据该变换可产生分布函数为 $F(x)$ 的随机数 x，其算法可用下列两个步骤实现：

(1) 产生均匀分布的随机数 u，即 $u\sim U(0,1)$；(2) 计算 $x=F^{-1}(u)$

2. 产生指数分布随机数的方法

指数分布的概率密度函数为

$$f(x)=\begin{cases}\dfrac{1}{\beta}e^{-\frac{x}{\beta}} & ,\quad x\geqslant 0\\ 0 & ,\quad \text{其它}\end{cases}$$

其分布函数为

$$F(x)=\begin{cases}1-e^{-\frac{x}{\beta}} & ,\quad x\geqslant 0\\ 0 & ,\quad \text{其它}\end{cases}$$

指数分布的均值为 β，方差为 β^2。

根据上述的逆变换法，产生指数分布随机数的方法为

(1) 产生均匀分布的随机数 u，即 $u\sim U(0,1)$；(2) 计算 $x=-\beta\ln(u)$

三、使用说明

1. 子函数语句

```
double exponent(beta,s)
```

2. 形参说明

beta ——双精度实型变量。指数分布的均值 β。

s ——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:exponent.c)

```
#include "math.h"
#include "uniform.c"
double exponent(beta,s)
double beta;
long int  *s;
{ double u,x;
  double uniform();
  u=uniform(0.0,1.0,s);
  x=-beta*log(u);
  return(x);
}
```

五、例　题

产生 50 个均值为 2、方差为 4 的指数分布的随机数。

主函数程序(文件名:exponent.m):

```
#include "stdio.h"
#include "exponent.c"
main()
{ int i,j; long int s;
  double x, beta;
  double exponent();
  beta=2.0; s=13579;
  for (i=0; i<10; i++)
   { for (j=0; j<5; j++)
       { x=exponent(beta,&s);
         printf("%13.7f",x);
       }
     printf("\n");
   }
}
```

运行结果:

```
1.4569871    0.0209201    0.6551459    0.5186231    0.2411175
0.6044725    1.0581442    0.4101700    0.4161992    0.4512997
0.2000017    1.8783014    1.3362513    1.7963731    0.1150597
0.7961070    1.4873781    4.5041685    4.7575350    1.3888938
```

3.5220201	1.4822608	1.0361474	3.9497085	0.4257289
0.2033371	2.3579595	7.2655635	5.1813278	2.3157520
1.6314957	1.3050398	0.2178681	0.1352042	2.2598519
0.8556571	3.3975954	6.5813570	4.0430121	0.2735539
1.1002899	0.7384279	0.8234798	0.4326338	3.1538658
1.3349979	5.4807363	0.0044701	1.6674272	1.7567765

§1.4 拉普拉斯(Laplace)分布的随机数

一、功　能

产生拉普拉斯分布的随机数。

二、方法简介

1. 产生随机变量的组合法

将分布函数 $F(x)$分解为若干个较简单的子分布函数的线性组合

$$F(x)=\sum_{i=1}^{K}p_iF_i(x)$$

其中 $p_i>0\ (\forall i)$，且 $\sum_{i=1}^{K}p_i=1$，$F_i(x)$ 是分布函数。

定理　若随机变量 ξ～离散分布$\{p_i\}$，即 $P(\xi=i)=p_i$，并且 $x\sim F_\xi(x)$，取 $z=x$，则 $z\sim F(x)=\sum_{i=1}^{K}p_iF_i(x)$

证明　z 的分布函数为

$$P(z\leqslant t)=P\left((z\leqslant t)\cap\bigcup_{i=1}^{K}(\xi=i)\right)=\sum_{i=1}^{K}P(z\leqslant t,\xi=i)$$

$$=\sum_{i=1}^{K}P(\xi=i)P(z\leqslant t|\xi=i)=\sum_{i=1}^{K}p_iF_i(t)=F(t)$$

定理证毕。

根据此定理，我们给出产生随机数的组合算法如下：

(1)产生一个正随机整数 ξ，使得 $P(\xi=i)=p_i\quad(i=1,2,\cdots,K)$

(2)在 $\xi=i$ 时，产生具有分布函数 $F_i(x)$的随机变量 x。

在该算法中，首先以概率 p_i 选择子分布函数 $F_i(x)$，然后取$F_i(x)$的随机数作为$F(x)$的随机数。

2. 产生拉普拉斯分布随机数的方法

拉普拉斯分布的概率密度函数为

$$f(x) = \frac{1}{2\beta} e^{-\frac{|x|}{\beta}}$$

拉普拉斯分布的均值为 0，方差为 $2\beta^2$。拉普拉斯分布也称为双指数分布。

根据上述的组合算法，产生拉普拉斯分布随机数的方法为

(1) 产生均匀分布的随机数 u_1 和 u_2，即 $u_1, u_2 \sim U(0,1)$；

(2) 计算 $x = \begin{cases} -\beta \ln(1-u_2) & u_1 \leqslant 0.5 \\ \beta \ln(u_2) & u_1 > 0.5 \end{cases}$

三、使用说明

1. 子函数语句

```
double laplace(beta,s)
```

2. 形参说明

beta ——双精度实型变量。拉普拉斯分布的参数 β。

s ——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:laplace.c)

```
#include "math.h"
#include "uniform.c"
double laplace(beta,s)
double beta;
long int *s;
{ double u1,u2,x;
  double uniform();
  u1=uniform(0.,1.,s);
  u2=uniform(0.,1.,s);
  if (u1<=0.5)
    x=-beta*log(1.-u2);
  else
    x=beta*log(u2);
  return(x);
}
```

五、例　题

产生 50 个参数 β 为 1.5 的拉普拉斯分布的随机数。

主函数程序(文件名:laplace.m):

```
#include "stdio.h"
```

```
#include "laplace.c"
main()
{ int i,j; long int s;
  double x, beta;
  double laplace();
  beta=1.5; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=laplace(beta,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果：

6.8481255	−0.3889673	−0.4533544	−0.3076275	−0.3384748
−1.4087260	−1.3472798	−0.5970803	0.1666987	1.0377737
0.9710420	−2.9622812	−0.1525028	0.0401976	0.5656504
1.1032428	−0.1014031	1.5829982	0.0569089	3.0855193
−0.5538209	−0.3244754	1.0792059	9.1569157	0.8053746
−0.6078966	−1.5382193	0.9028431	−3.4637606	0.2715135
−0.0907091	−0.9236331	−1.0849801	5.9835591	0.7128560
−1.7257895	0.1008553	−0.9126053	−0.9336386	1.3214850
−3.7663898	0.0510632	0.0513326	−0.1540588	1.1192728
0.7144444	−2.1629963	−0.8393618	−2.9372783	−2.0191078

§1.5 瑞利(Rayleigh)分布的随机数

一、功 能

产生瑞利分布的随机数。

二、方法简介

瑞利分布的概率密度函数为

$$f(x)=\frac{x}{\sigma^2}\mathrm{e}^{-x^2/2\sigma^2}\qquad x>0$$

瑞利分布的均值为 $\sigma\sqrt{\frac{\pi}{2}}$，方差为 $\left(2-\frac{\pi}{2}\right)\sigma^2$。

首先用逆变换法产生参数 $\beta=2$ 的指数分布的随机变量 y，其概率密度函数为 $f(y)=\frac{1}{2}e^{-\frac{y}{2}}$；然后通过变换 $x=\sigma\sqrt{y}$，产生瑞利分布的随机变量 x。具体方法如下：

(1) 产生均匀分布的随机数 u，即 $u\sim U(0,1)$；

(2) 计算 $y=-2\ln(u)$

(3) 计算 $x=\sigma\sqrt{y}$

三、使用说明

1. 子函数语句

```
double rayleigh(sigma,s)
```

2. 形参说明

sigma ——双精度实型变量。瑞利分布的参数 σ。

s ——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:rayleigh.c)

```
#include "math.h"
#include "uniform.c"
double rayleigh(sigma,s)
double sigma;
long int *s;
{ double u,x;
  double uniform();
  u=uniform(0.,1.,s);
  x=-2.0*log(u);
  x=sigma*sqrt(x);
  return(x);
}
```

五、例　题

产生 50 个参数 $\sigma=1$ 的瑞利分布的随机数。

主函数程序(文件名:rayleigh.m)：

```
#include "stdio.h"
#include "rayleigh.c"
main()
{ int i,j; long int s;
  double x, sigma;
```

```
    double rayleigh();
    sigma=1.0; s=13579;
    for (i=0;i<10;i++)
      { for (j=0;j<5;j++)
          { x=rayleigh(sigma,&s);
            printf("%13.7f",x);
          }
        printf("\n");
      }
  }
```

运行结果：

1.2070572	0.1446379	0.8094109	0.7201549	0.4910371
0.7774783	1.0286614	0.6404452	0.6451350	0.6717884
0.4472155	1.3705114	1.1559633	1.3402884	0.3392045
0.8922483	1.2195811	2.1223025	2.1811774	1.1785134
1.8767046	1.2174813	1.0179132	1.9873873	0.6524790
0.4509292	1.5355649	2.6954708	2.2762530	1.5217595
1.2773002	1.1423835	0.4667634	0.3677012	1.5032804
0.9250173	1.8432568	2.5654154	2.0107243	0.5230238
1.0489471	0.8593183	0.9074579	0.6577490	1.7759126
1.1554211	2.3410971	0.0668588	1.2912890	1.3254344

§1.6 对数正态分布的随机数

一、功　能

产生对数正态分布的随机数。

二、方法简介

对数正态分布的概率密度函数为

$$f(x)=\begin{cases}\dfrac{1}{x\sqrt{2\pi}\sigma}\exp\left(-\dfrac{(\ln x-\mu)^2}{2\sigma^2}\right) & x>0\\ 0 & x\leqslant 0\end{cases}$$

对数正态分布的均值为 $e^{\mu+\sigma^2/2}$，方差为$(e^{\sigma^2}-1)e^{2\mu+\sigma^2}$。

首先产生正态分布的随机变量 y，然后通过变换 $x=e^y$ 产生对数正态分布的随机变量 x。具体方法如下：

(1) 产生正态分布的随机数 y，即 $y\sim N(\mu,\sigma)$；

(2) 计算 $x=e^y$

三、使用说明

1. 子函数语句

```
double lognorm(u,sigma,s)
```

2. 形参说明

u ——双精度实型变量。对数正态分布的参数 μ。
sigma ——双精度实型变量。对数正态分布的参数 σ。
s ——长整型指针变量，*s 为随机数的种子。

四、子函数程序(文件名:lognorm.c)

```
#include "math.h"
#include "gauss.c"
double lognorm(u,sigma,s)
double u,sigma;
long int *s;
{ double x,y;
  double gauss();
  y=gauss(u,sigma,s);
  x=exp(y);
  return(x);
}
```

五、例 题

产生 50 个参数 $\mu=0$、$\sigma=0.5$ 的对数正态分布的随机数。

主函数程序(文件名:lognorm.m):

```
#include "stdio.h"
#include "lognorm.c"
main()
{ int i,j; long int s;
  double x, u, sigma;
  double lognorm();
  u=0.0; sigma=0.5; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=lognorm(u,sigma,&s);
          printf("%13.7f",x);
```

```
        }
        printf("\n");
    }
}
```

运行结果：

4.2625203	0.6348105	1.1074915	0.8793188	0.6532200
1.4916008	0.6105892	1.0218035	0.3829900	1.1356161
0.8315511	1.8354563	0.5904593	2.3456149	0.4290005
0.7815813	2.7124009	0.7415348	1.9082086	1.0891325
0.7705273	0.8166317	1.5188051	0.7616608	2.2764161
1.3059697	0.6662967	0.8233758	0.8804533	1.2642837
0.8181845	0.9456485	0.6154250	1.3841861	1.9328243
1.3075050	1.4524198	1.9420923	0.8170368	2.4661858
0.6529210	0.8829085	2.3190839	1.6044350	1.2724800
1.7953745	0.9173305	1.1089542	0.3678910	1.5189326

§1.7　柯西(Cauchy)分布的随机数

一、功　能

产生柯西分布的随机数。

二、方法简介

柯西分布的概率密度函数为

$$f(x) = \frac{\beta}{\pi[\beta^2 + (x-\alpha)^2]} \qquad \beta > 0$$

通常用 $C(\alpha,\beta)$表示，其分布函数为

$$F(x) = \frac{1}{2} + \frac{1}{\pi}\mathrm{arctg}\left(\frac{x-\alpha}{\beta}\right)$$

用逆变换法产生柯西分布 $C(\alpha,\beta)$随机变量 x，其具体方法如下：

(1)产生均匀分布的随机数 u，即 $u \sim U(0,1)$；

(2) 计算 $x = \alpha - \dfrac{\beta}{\mathrm{tg}(u\pi)}$

三、使用说明

1. 子函数语句

```
double cauchy(a,b,s)
```

2. 形参说明

a——双精度实型变量。柯西分布的参数 α。

b——双精度实型变量。柯西分布的参数 β。

s——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:cauchy.c)

```
#include "math.h"
#include "uniform.c"
double cauchy(a,b,s)
double a,b;
long int *s;
{ double u,x;
  double uniform();
  u=uniform(0.0,1.0,s);
  x=a-b/tan(3.1415926*u);
  return(x);
}
```

五、例　题

产生 50 个参数 $\alpha=1$、$\beta=1$ 的柯西分布的随机数。

主函数程序(文件名:cauchy.m):

```
#include "stdio.h"
#include "cauchy.c"
main()
{ int i,j; long int s;
  double x, alpha, beta;
  double cauchy();
  alpha=1.0; beta=1.0; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=cauchy(alpha,beta,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果:

0.9453936	31.5794754	1.8308272	2.1457105	3.6826832
1.9341286	1.2876379	2.5179656	2.4928074	2.3583286
4.2446246	0.6433801	1.0398206	0.7002780	6.6349649
1.5983145	0.9224268	−1.9153867	−2.3374801	0.9979597
−0.6684446	0.9262747	1.3099350	−1.1463964	2.4543467
4.1908474	0.3092863	−11.0100975	−3.1670158	0.3393057
0.8167456	1.0652235	3.9752383	5.8008814	0.3787720
1.5171593	−0.5444290	−7.5111265	−1.2628045	3.3546693
1.2462881	1.6854823	1.5600127	2.4274268	−0.3179573
1.0408319	−3.8636963	143.5733032	0.7910515	0.7279566

§1.8 韦伯(Weibull)分布的随机数

一、功　能

产生韦伯分布的随机数。

二、方法简介

韦伯分布的概率密度函数为

$$f(x)=\begin{cases}\dfrac{\alpha}{\beta^{\alpha}}x^{\alpha-1}\mathrm{e}^{-\left(\frac{x}{\beta}\right)^{\alpha}} & x\geqslant 0,\alpha>0,\beta>0\\ 0 & x<0\end{cases}$$

用 $W(\alpha,\beta)$表示，其分布函数为

$$F(x)=\begin{cases}1-\mathrm{e}^{-\left(\frac{x}{\beta}\right)^{\alpha}} & x\geqslant 0,\ \alpha>0,\ \beta>0\\ 0 & x<0\end{cases}$$

韦伯分布的均值为 $\dfrac{\beta}{\alpha}\Gamma\left(\dfrac{1}{\alpha}\right)$ 。

应用逆变换方法，我们得到产生韦伯分布随机变量 x 的算法如下：

(1) 产生均匀分布的随机数 u,即 $u\sim U(0,1)$；

(2) 计算 $x=\beta(-\ln(u))^{1/\alpha}$

三、使用说明

1. 子函数语句

```
double weibull(a,b,s)
```

2. 形参说明

a——双精度实型变量。韦伯分布的参数 α。

b——双精度实型变量。韦伯分布的参数 β。

s —长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:weibull.c)

```
#include "math.h"
#include "uniform.c"
double weibull(a,b,s)
double a,b;
long int  *s;
{ double u,x;
  double uniform();
  u=uniform(0.0,1.0,s);
  u=-log(u);
  x=b*pow(u,1.0/a);
  return(x);
}
```

五、例　题

产生 50 个参数 $\alpha=2$、$\beta=1$ 的韦伯分布的随机数。

主函数程序(文件名:weibull.m):

```
#include "stdio.h"
#include "weibull.c"
main()
{ int i,j; long int s;
  double x, alpha, beta;
  double weibull();
  alpha=2.0; beta=1.0; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=weibull(alpha,beta,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
0.8535183    0.1022744    0.5723399    0.5092264    0.3472157
0.5497602    0.7273734    0.4528631    0.4561793    0.4750262
0.3162291    0.9690979    0.8173895    0.9477270    0.2398538
```

0.6309148	0.8623741	1.5006946	1.5423253	0.8333348
1.3270305	0.8608893	0.7197734	1.4052951	0.4613724
0.3188551	1.0858084	1.9059857	1.6095541	1.0760465
0.9031876	0.8077870	0.3300516	0.2600040	1.0629798
0.6540860	1.3033794	1.8140227	1.4217968	0.3698337
0.7417176	0.6076298	0.6416696	0.4660988	1.2557600
0.8170061	1.6554056	0.0472763	0.9130792	0.9372237

§1.9 爱尔朗(Erlang)分布的随机数

一、功　能

产生爱尔朗分布的随机数。

二、方法简介

爱尔朗分布的概率密度函数为

$$f(x)=\begin{cases}\dfrac{\beta^{-m}x^{m-1}}{(m-1)!}\,e^{-\frac{x}{\beta}} & x\geqslant 0,\quad \beta>0\\ & x<0\end{cases}$$

通常用 $E(m,\beta)$ 表示。爱尔朗分布的均值为 $m\beta$，方差为 $m\beta^2$。显然，当 $m=1$ 时，$E(1,\beta)$ 就是参数为 β 的指数分布的概率密度函数。

若 $y_i(i=1,2,\cdots,m)$ 是独立同分布(IID)的参数为 β 的指数随机变量，则 $x=\sum_{i=1}^{m}y_i$ 服从爱尔朗分布 $E(m,\beta)$。因此，先用逆变换法产生指数分布的随机变量 $y_i(y_i=-\beta\ln(u_i),u_i\sim U(0,1))$，然后产生爱尔朗分布的随机变量 x，即

$$x=\sum_{i=1}^{m}y_i=\sum_{i=1}^{m}(-\beta\ln(u_i))=-\beta\ln\left(\prod_{i=1}^{m}u_i\right)$$

产生爱尔朗分布随机变量 x 的具体算法如下：

(1) 产生 IID 均匀分布的随机数 $u_1,u_2,\cdots,u_m$，即 $u_i\sim U(0,1)$；

(2) 计算 $x=-\beta\ln\left(\prod_{i=1}^{m}u_i\right)$

三、使用说明

1. 子函数语句

```
double erlang(m,beta,s)
```

2. 形参说明

m ——整型变量。爱尔朗分布的参数 m。

beta ——双精度实型变量。爱尔朗分布的参数 β。

s ——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:erlang.c)

```
#include "math.h"
#include "uniform.c"
double erlang(m,beta,s)
int m;
double beta;
long int *s;
{ int i; double u,x;
  double uniform();
  for (u=1.0,i=0; i<m; i++)
  u*=uniform(0.0,1.0,s);
  x=-beta*log(u);
  return(x);
}
```

五、例 题

产生 50 个参数 $m=3$、$\beta=1$ 的爱尔朗分布的随机数。

主函数程序(文件名:erlang.m):

```
#include "stdio.h"
#include "erlang.c"
main()
{ int i,j,m; long int s;
  double x, beta;
  double erlang();
  m=3; beta=1.0; s=13579;
  for(i=0;i<10;i++)
    { for(j=0;j<5;j++)
        { x=erlang(m,beta,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
1.0665267    0.6821066    0.9422567    1.2648014    1.6238420
3.3938267    4.8342242    3.2340584    1.4935127    7.3813219
```

1.5772018	1.6253566	7.0109820	1.0561359	2.2049897
3.4101019	2.2240145	1.9299352	1.9592353	5.9587183
0.5812231	1.5929523	4.2617917	2.7262666	4.4612184
1.3056705	1.8735027	9.1363468	4.4265428	1.6808789
2.2389832	2.5986791	2.9839444	4.1416554	3.6086712
4.1421738	2.3656099	1.2727489	4.0904193	7.5296683
4.3179040	2.9108481	2.7557034	2.3836980	1.6895797
2.4720519	2.0467947	1.7919904	1.5726769	2.3995914

§1.10 贝努里(Bernoulli)分布的随机数

一、功　能

产生贝努里分布的随机数。

二、方法简介

贝努里分布的概率函数为

$$f(x) = \begin{cases} p\ , & x = 1 \\ 1 - p, & x = 0 \end{cases}$$

用 BN(p)表示。贝努里分布的均值为 p，方差为 $p(1-p)$。

产生贝努里分布随机变量 x 的算法如下：

(1) 产生均匀分布的随机数 u，即 $u \sim U(0,1)$；

(2) 如果 $u \leqslant p$，那么 $x=1$；否则，$x=0$

三、使用说明

1. 子函数语句

```
int bn(p,s)
```

2. 形参说明

p——双精度实型变量。贝努里分布的参数 p。

s——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:bn.c)

```
#include "uniform.c"
int bn(p,s)
double p;
long int *s;
```

```
{ int x; double u;
  double uniform();
  u=uniform(0.0,1.0,s);
  x=(u<=p)? 1:0;
  return(x);
}
```

五、例　题

产生 50 个参数 $p=0.7$ 的贝努里分布的随机数。

主函数程序(文件名:bn.m):

```
#include "stdio.h"
#include "bn.c"
main()
{ int i,j,x; long int s;
  double p;
  p=0.7; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=bn(p,&s);
          printf("%11d",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
          1          0          0          0          0
          0          1          0          0          0
          0          1          1          1          0
          1          1          1          1          1
          1          1          1          1          0
          0          1          1          1          1
          1          1          0          0          1
          1          1          1          1          0
          1          1          1          0          1
          1          1          0          1          1
```

§1.11 贝努里-高斯分布的随机数

一、功 能

产生贝努里-高斯分布的随机数。

二、方法简介

贝努里-高斯分布的随机变量 x 是贝努里分布的随机变量 y 与高斯分布的随机变量 z 的乘积，即 $x=y\times z$。因此，贝努里-高斯分布的随机数可视为：每当贝努里序列中有1出现时，打开高斯随机数发生器，并用其输出代替1。贝努里-高斯分布的均值为 $p\mu$，方差为 p，其中 p 是贝努里分布的参数，μ 是高斯分布的均值。

在地震勘探信号处理中，常用贝努里-高斯序列描述地下主要层状结构的反射作用。

产生贝努里-高斯分布随机变量 x 的算法如下：

(1) 产生贝努里分布的随机数 y，即 $y\sim \mathrm{BN}(p)$；

(2) 产生高斯分布的随机数 z，即 $z\sim \mathrm{N}(\mu,\sigma)$；

(2) 计算 $x=y\times z$

三、使用说明

1. 子函数语句

```
int bg(p,mean,sigma,s)
```

2. 形参说明

p ——双精度实型变量。贝努里分布的参数 p。

mean ——双精度实型变量。高斯分布的均值 μ。

sigma ——双精度实型变量。高斯分布的均方差 σ。

s ——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:bg.c)

```
#include"gauss.c"
double bg(p,mean,sigma,s)
double p,mean,sigma;
long int *s;
{ double u,x;
  double uniform(), gauss();
  u=uniform(0.0,1.0,s);
  if (u<=p)
```

```
      x=gauss(mean,sigma,s);
    else
      x=0.0;
    return(x);
}
```

五、例　题

产生 50 个参数 $p=0.4$、$\mu=0$、$\sigma=1$ 的贝努里-高斯分布的随机数。

主函数程序(文件名:bg.m):

```
#include "stdio.h"
#include "bg.c"
main()
{ int i,j; long int s;
  double p, x, mean, sigma;
  double bg(double,double,double,long int *);
  p=0.4; mean=0.0; sigma=1.0; s=12357;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=bg(p,mean,sigma,&s);
          printf("%13.7f",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
   0.6910229    0.0000000    1.6546841    0.0000000    1.4901104
   0.0000000    0.0000000    0.0000000    0.3757915    0.0000000
  -1.0547123    0.0000000    0.0000000    0.0000000    1.1713009
   0.0000000    0.0000000    0.0000000    0.0000000    0.0000000
   0.4781475    0.0000000    0.0000000    0.0000000    0.1607037
   0.9864216    0.0000000    0.7719517    0.0000000    0.0000000
   0.0000000    0.0000000    0.0000000    0.0000000   -0.2416019
   0.8279819    0.0000000    2.7472401   -0.6780338    0.0000000
   0.1726513    0.1959400   -0.1427860    0.2121983   -1.9189777
   0.0000000    0.0000000    0.0000000    0.0000000   -0.9137707
```

§ 1.12 二项式分布的随机数

一、功 能

产生二项式分布的随机数。

二、方法简介

二项式分布的概率函数为

$$f(x) = C_n^x p^x (1-p)^{n-x} \qquad x \in \{0,1,\cdots,n\}$$

用 Bin (n,p) 表示。二项式分布的均值为 np，方差为 $np(1-p)$。当 $n=1$ 时，Bin(n,p)就是贝努里分布 BN(p)。

若 $y_i(i=1,2,\cdots,n)$是独立同分布(IID)的参数为 p 的贝努里分布随机变量，则 $x=\sum_{i=1}^{m} y_i$ 服从二项式分布 Bin(n,p)。因此，产生二项式分布随机变量 x 的算法如下：

(1)产生 IID 贝努里分布的随机数 $y_1,y_2,\cdots,y_n$，即 $y_i \sim$ BN(p)；

(2) 计算 $x=\sum_{i=1}^{n} y_i$

三、使用说明

1. 子函数语句

```
int bin(n,p,s)
```

2. 形参说明

n——整型变量。二项式分布的参数 n。

p——双精度实型变量。二项式分布的参数 p。

s——长整型指针变量。 *s 为随机数的种子。

四、子函数程序(文件名:bin.c)

```
#include "bn.c"
int bin(n,p,s)
int n;
double p;
long int *s;
{ int i; double x;
  int bn();
  for (x=0.0,i=0; i<n; i++)
```

```
    x+=bn(p,s);
  return(x);
}
```

五、例　题

产生 50 个参数 $n=5$、$p=0.7$ 的二项式分布的随机数。

主函数程序(文件名:bin.m):

```
#include "stdio.h"
#include "bin.c"
main()
{ int i,j,n,x; long int s;
  double p;
  int bin();
  n=5; p=0.7; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=bin(n,p,&s);
          printf("%11d",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
          1          1          3          5          4
          4          3          4          4          4
          5          4          2          4          4
          4          5          3          4          5
          5          4          3          4          5
          4          2          3          4          3
          5          3          3          4          4
          3          3          5          2          2
          2          4          4          4          2
          3          4          3          4          3
```

§1.13 泊松(Poisson)分布的随机数

一、功　能

产生泊松分布的随机数。

二、方法简介

泊松分布的概率函数为

$$f(x)=\frac{\lambda^x e^{-\lambda}}{x!} \qquad x\in\{0,1,\cdots,\lambda\},\lambda>0$$

用 P(λ)表示。泊松分布的均值为 λ，方差为 λ。

定理　若 $\lambda>0$，x 是整数，u_i 是(0,1)区间上均匀分布的随机数，即 $u_i\sim U(0,1)$，且有

$$\prod_{i=0}^{x}u_i\geqslant e^{-\lambda}>\prod_{i=0}^{x+1}u_i$$

那么 x 是一个以 λ 为均值的泊松分布随机变量。

产生泊松分布随机变量 x 的算法如下：

(1) 设 $b=1$，$i=0$；

(2) 产生均匀分布的随机数 u_i，即 $u_i\sim U(0,1)$；

(3) 计算 $b\leftarrow bu_i$；

(4) 如果 $b\geqslant e^{-\lambda}$，那么 $i\leftarrow i+1$，返回到(2)；

(5) 取 $x=i$

三、使用说明

1. 子函数语句

```
int poisson(lambda,s)
```

2. 形参说明

lambda ——双精度实型变量。泊松分布的均值 λ。

s　——长整型指针变量。*s 为随机数的种子。

四、子函数程序(文件名:poisson.c)

```
#include "uniform.c"
#include "math.h"
int poisson(lambda,s)
double lambda;
```

```
long int  *s;
{ int i,x; double a,b,u;
  double uniform();
  a=exp(-lambda);
  i=0;
  b=1.0;
  do
    { u=uniform(0.0,1.0,s);
      b *=u;
      i++;
    } while (b>=a);
  x=i-1;
  return(x);
}
```

五、例　题

产生 50 个均值 $\lambda=4$ 的泊松分布的随机数。

主函数程序(文件名:poisson.m):

```
#include "stdio.h"
#include "poisson.c"
main()
{ int i,j,x; long int s;
  double n;
  int bin();
  n=4.0; s=13579;
  for (i=0;i<10;i++)
    { for (j=0;j<5;j++)
        { x=poisson(n,&s);
          printf("%11d",x);
        }
      printf("\n");
    }
}
```

运行结果:

```
   12     4     2     5     1
    6     1     6     3     5
    3     7     4     1     7
    1     1     5     5     3
```

3	2	6	1	3
3	4	6	6	5
2	1	7	2	2
3	0	5	4	3
4	7	4	3	4
1	7	3	6	4

§1.14 ARMA(p,q)模型数据的产生

一、功　能

产生自回归滑动平均模型 $ARMA(p,q)$的数据。

二、方法简介

自回归滑动平均模型 ARMA(p,q)为

$$x(n)+\sum_{i=1}^{p}a_i x(n-i)=\sum_{i=0}^{q}b_i w(n-i)$$

其中 $a_i(i=1,2,\cdots,p)$是自回归系数，$b_i(i=0,1,\cdots,q)$是滑动平均系数，$w(n)$是白噪声。

给定白噪声 $w(n)$的均值和方差，便可以由上式产生 ARMA(p,q)的数据。

三、使用说明

1. 子函数语句

```
void arma(a,b,p,q,mean,sigma,seed,x,n)
```

2. 形参说明

a ——双精度实型一组数组，长度为(p+1)。ARMA(p,q)模型的自回归系数。
b ——双精度实型一组数组，长度为(q+1)。ARMA(p,q)模型的滑动平均系数。
p ——整型变量。ARMA(p,q)模型的自回归阶数。
q ——整型变量。ARMA(p,q)模型的滑动平均阶数。
mean ——双精度实型变量。产生白噪声所用的正态分布的均值 μ。
sigma——双精度实型变量。产生白噪声所用的正态分布的均方差 σ。
seed ——长整型指针变量。*seed 为随机数的种子。
x ——双精度实型一组数组，长度为 n。存放 ARMA(p,q)模型的数据。
n ——整型变量。ARMA(p,q)模型数据的长度。

四、子函数程序(文件名:arma.c)

```
#include "stdlib.h"
```

```
#include "gauss.c"
void arma(a,b,p,q,mean,sigma,seed,x,n)
int n,p,q;
long *seed;
double mean,sigma,a[],b[],x[];
{ int i,k,m;
  double s,*w;
  w = malloc(n*sizeof(double));
  for (k=0;k<n;k++)
    { w[k] = gauss(mean,sigma,seed); }
  x[0] = b[0]*w[0];
  for (k=1;k<=p;k++)
    { s = 0.0;
      for (i=1;i<=k;i++)
        { s += a[i]*x[k-i]; }
      s = b[0]*w[k] - s;
      if (q == 0)
        { x[k] = s;
          continue;
        }
      m = (k > q) ? q : k;
      for (i=1;i<=m;i++)
        { s += b[i]*w[k-i]; }
      x[k] = s;
    }
  for (k=(p+1);k<n;k++)
    { s = 0.0;
      for (i=1;i<=p;i++)
        { s += a[i]*x[k-i]; }
      s = b[0]*w[k] - s;
      if ( q == 0 )
        { x[k] = s;
          continue;
        }
      for (i=1;i<=q;i++)
        { s += b[i]*w[k-i]; }
      x[k] = s;
    }
```

```
  free(w);
}
```

五、例　题

ARMA(2,2)模型为

$$x(k)+1.45x(k-1)+0.6x(k-2)=w(k)-0.2w(k)-0.1w(k)$$

其中 $w(k)$是均值为 0、均方差为 0.5 的高斯白噪声。产生 200 个数据，并将其存于数据文件 arma.dat 中。

主函数程序(文件名:arma.m):

```
#include "stdio.h"
#include "arma.c"
main()
{ int i,n,p,q;
  long seed;
  double mean,sigma,x[200];
  static double a[3] = {1.0, 1.45, 0.6};
  static double b[3] = {1.0, -0.2, -0.1,};
  FILE *fp;
  n = 200;
  p = 2;
  q = 2;
  seed = 13579l;
  mean = 0.0;
  sigma = 0.5;
  arma(a,b,p,q,mean,sigma,&seed,x,n);
  for (i=0;i<32;i+=4)
    { printf("        %10.7lf        %10.7lf",x[i],x[i+1]);
      printf("        %10.7lf        %10.7lf",x[i+2],x[i+3]);
      printf("\n");
    }
  fp = fopen("arma.dat","w");
  for (i=0;i<n;i++)
    fprintf(fp,"%d %lf\n",i,x[i]);
  fclose(fp);
}
```

运行结果：

ARMA 模型的前 32 个数据为

1.4498606　　-2.8466986　　3.3057938　　-3.1889663

2.2301953	−0.8225244	−0.6761735	1.5542167
−2.7626373	3.3902617	−3.3722200	3.4870302
−3.6627286	4.1159276	−4.7345792	4.2331519
−2.1655715	0.1262312	1.7224958	−2.5873058
2.3757229	−2.0513840	2.0335784	−2.0534403
2.5926021	−2.3975581	1.3792284	−0.7011795
0.1413254	0.4951895	−1.0376567	1.1682870

200 点 ARMA(2,2)数据如图 1−1−1 所示。

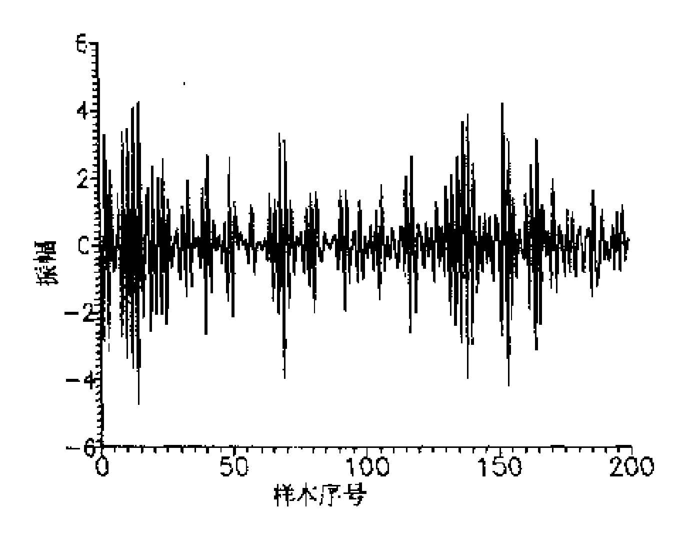


图 1—1—1　ARMA(2,2)模型的序列

§1.15　含有高斯白噪声的正弦组合信号的产生

一、功　能

产生含有高斯白噪声的正弦组合信号。

二、方法简介

含有高斯白噪声的 M 个正弦信号的组合为

$$x(n) = \sum_{i=1}^{M} A_i \sin(2\pi f_i \Delta T\ n + \theta_i) + N(0,\sigma^2)$$

其中 A_i、f_i 和 $\theta_i(i=1,2,\cdots,M)$分别是第 i 个正弦信号的振幅、频率和相位。$\Delta T=1/f_s$，ΔT 是采样间隔，f_s 是采样频率(以赫兹为单位)。$N(0,\sigma^2)$是高斯白噪声，它的均值为零，方差为 σ^2。

三、使用说明

1. 子函数语句

```
void sinwn(a,f,ph,m,fs,snr,seed,x,n)
```

2. 形参说明

a ——双精度实型一组数组，长度为 m。各正弦信号的振幅。
f ——双精度实型一组数组，长度为 m。各正弦信号的频率。
ph ——双精度实型一组数组，长度为 m。各正弦信号的相位。
m ——整型变量。正弦信号的个数。
fs ——双精度实型变量。采样频率(用赫兹表示)。
snr ——双精度实型变量。信噪比(用 dB 表示)。
seed ——长整型变量。随机数的种子。
x ——双精度实型一组数组，长度为 n。存放所产生的数据。
n ——整型变量。数据长度。

四、子函数程序(文件名:sinwn.c)

```
#include "math.h"
#include "gauss.c"
void sinwn(a,f,ph,m,fs,snr,seed,x,n)
int m,n;
long seed;
double fs,snr,a[],f[],ph[],x[];
{ int i,k;
  double z,pi,nsr;
  pi = 4.0*atan(1.0);
  z = snr/10.0;
  z = pow(10.0,z);
  z = 1.0/(2*z);
  nsr = sqrt(z);
  for (i=0;i<m;i++)
    { f[i] = 2*pi*f[i]/fs;
      ph[i] = ph[i]*pi/180.0;
    }
  for (k=0;k<n;k++)
    { x[k] = 0.0;
      for (i=0;i<m;i++)
        { x[k] = x[k] + a[i]*sin(k*f[i] + ph[i]); }
      x[k] = x[k] + nsr*gauss(0.0,1.0,&seed);
      }
}
```

五、例　题

例 1：单正弦信号的振幅为 1，频率为 5 Hz，相位为 45 度，无噪声干扰。产生 200 个数据，并将其存于数据文件 sinwn1.dat 中。

主函数程序（文件名：sinwn1.m）：

```
#include "stdio.h"
#include "sinwn.c"
main()
{ int i,m,n;
  long seed;
  double fs,snr,x[200];
  static double a[1] = { 1 };
  static double f[1] = { 5 };
  static double ph[1] = { 45 };
  FILE *fp;
  m = 1;
  n = 200;
  seed = 13579l;
  fs = 150;
  snr = 1000.0;
  sinwn(a,f,ph,m,fs,snr,seed,x,n);
  printf("\n Single Sinusoidal Signal\n");
  for (i=0;i<32;i++)
    { printf("        %10.7lf",x[i]);
      if ( i%4 == 3 ) printf("\n");
    }
  fp = fopen("sinwn1.dat","w");
  for (i=0;i<n;i++)
    { fprintf(fp,"%3d        %12.7lf\n",i,x[i]);}
  fclose(fp);
}
```

运行结果：

正弦信号的前 32 个数据为

0.7071068	0.8386706	0.9335804	0.9876883
0.9986295	0.9659258	0.8910065	0.7771460
0.6293204	0.4539905	0.2588190	0.0523360
−0.1564345	−0.3583679	−0.5446390	−0.7071068
−0.8386706	−0.9335804	−0.9876883	−0.9986295

−0.9659258	−0.8910065	−0.7771460	−0.6293204
−0.4539905	−0.2588190	−0.0523360	0.1564345
0.3583679	0.5446390	0.7071068	0.8386706

200 点正弦信号如图 1−1−2 所示。

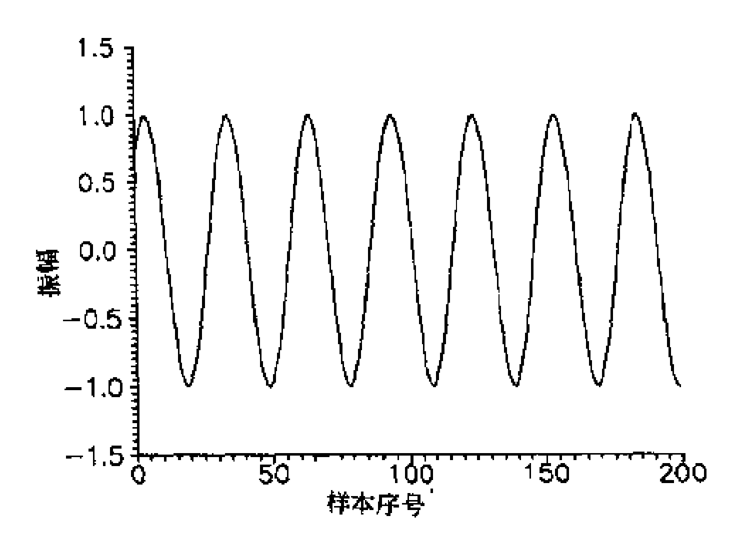


图 1−1−2　正弦序列

例 2：三个正弦信号的振幅为 $A_1=A_2=A_3=1$，频率为 $f_1=10$，$f_2=17$，$f_3=50$，相位为 $\theta_1=45$ 度，$\theta_2=10$ 度，$\theta_3=88$ 度，采样频率为 150 Hz。该正弦组合信号受到高斯白噪声 $N(0,\sigma^2)$的干扰，信噪比 SNR = 5 dB。产生 200 个数据，并将其存于数据文件 sinwn2.dat 中。

主函数程序(文件名：sinwn2.m)：

```
#include "stdio.h"
#include "sinwn.c"
main()
{ int i,m,n;
  long seed;
  double fs,snr,x[200];
  static double a[3] = {1, 1, 1 };
  static double f[3] = {10, 17, 50};
  static double ph[3] = {45, 10, 88};
  FILE *fp;
  m = 3;
  n = 200;
  seed = 13579l;
  fs = 150;
```

```
    snr = 5.0;
    sinwn(a,f,ph,m,fs,snr,seed,x,n);
    printf("\n Three Sinusoidal Signals plus Gauss White Noise\n");
    for (i=0;i<32;i-+)
      { printf("       %10.7lf",x[i]);
        if ( i%4 == 3 ) printf("\n");
      }
    fp = fopen("sinwn2.dat","w");
    for (i=0;i<n;i++)
      { fprintf(fp,"%3d      %12.7lf\n",i,x[i]);}
    fclose(fp);
  }
```

运行结果：

含有高斯白噪声的正弦组合信号前 32 个数据为

```
 3.0331775    0.8776595    1.5495155    2.5265747
-0.0604063   -0.5123042   -0.5143918   -1.8987897
-2.5321971    0.4085494   -0.7340470    0.1641256
 0.7745283    0.1492174   -1.5032762    0.5202461
 0.4170906   -0.0514435    2.8173778    1.1355277
 0.4828501    1.2294239   -0.8140043   -2.3329036
-0.3340085   -1.9891476   -1.7898424    0.9148312
 0.3301306    0.9340795    1.9852839    0.1642199
```

200 点含有高斯白噪声的正弦组合信号如图 1—1—3 所示。

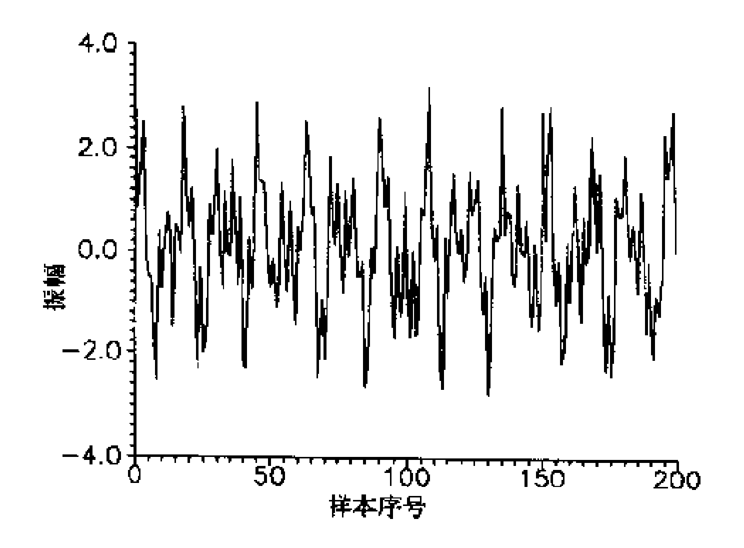


图 1—1—3　含有高斯白噪声的正弦组合序列

§1.16 解析信号的产生

一、功　能

用快速傅立叶变换技术产生解析信号。

二、方法简介

实信号 $x(n)$ 的解析信号可表示为

$$z(n) = x(n) + j\hat{x}(n)$$

其中 $\hat{x}(n)$ 是 $x(n)$ 的离散希尔伯特变换，它可用下式来表示

$$\hat{x}(n) = x(n) * h(n)$$

这里 $h(n)$ 是希尔伯特变换的单位冲激响应，即

$$h(n) = \begin{cases} \dfrac{2}{n\pi}\sin^2\left(\dfrac{n\pi}{2}\right), & n \neq 0 \\ 0, & n = 0 \end{cases}$$

设信号 $x(n)$ 的频谱为 $X(\omega)$，解析信号 $z(n)$ 的频谱为 $Z(\omega)$，则有

$$Z(\omega) = \begin{cases} 2X(\omega) & , \quad \omega > 0 \\ X(\omega) & , \quad \omega = 0 \\ 0 & , \quad \omega < 0 \end{cases}$$

因此，计算解析信号的步骤如下：

1. 计算 $x(n)$ 的 N 点 FFT，得到 $X(k)(k=0,1,\cdots,N-1)$。
2. 构造 $Z(k)$

$$Z(k) = \begin{cases} X(k) & , \quad k = 0 \\ 2X(k) & , \quad k = 1,2,\cdots,\dfrac{N}{2}-1 \\ 0 & , \quad \text{其它} \end{cases}$$

3. 计算 $Z(k)$ 的快速傅立叶反变换，从而得到解析信号 $z(n)$。

三、使用说明

1. 子函数语句

```
void analytic(x,y,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放实数输入信号，最后存放解析信号的实部(实际上，它没有变化，与原来完全相同)。

y ——双精度实型一维数组，长度为 n。存放解析信号的虚部。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:analytic.c)

```
#include "fft.c"
void analytic(x,y,n)
int n;
double x[],y[];
{ int i,n1;
  n1 = n/2;
  for (i=0;i<n;i++)
    { y[i] = 0.0; }
  fft(x,y,n,1);
  for(i=1;i<n1;i++)
    { x[i] = 2 * x[i];
      y[i] = 2 * y[i];
    }
  for (i=n1;i<n;i++)
    { x[i] = 0.0;
      y[i] = 0.0;
    }
  fft(x,y,n,-1);
}
```

五、例　题

设序列 $x(i)$为

$$x(i) = \sin(2\pi i/n)\ ,i = 0,1,\cdots,n-1$$

它的希尔伯特变换为

$$\hat{x}(i) = -\cos(2\pi i/n)\ ,i = 0,1,\cdots,n-1$$

因此，$x(i)$的解析信号为

$$z(i) = \sin(2\pi i/n) - j\cos(2\pi i/n)\ ,i = 0,1,\cdots,n-1$$

计算序列 $x(i)$的解析信号 $z(i)$,并与理想值进行比较。

主函数程序(文件名:analytic.m):

```
#include "stdio.h"
#include "math.h"
#include "analytic.c"
main()
{ int i,n;
  double x[64],y[64],z[64];
  void analytic();
```

```
n = 64;
for (i=0;i<n;i++)
  { x[i] = sin(2 * 3.14159265 * i/n); }
printf("\n Original Signal\n");
for (i=0;i<n/2;i+=4)
  { printf("        %10.7f        %10.7f",x[i],x[i+1]);
    printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
  }
for (i=0;i<n;i++)
  { z[i] = -cos(2 * 3.14159265 * i/n); }
printf("\n Ideal Discrete Hilbert Transform\n");
for (i=0;i<n/2;i+=4)
  { printf("        %10.7f        %10.7f",z[i],z[i+1]);
    printf("        %10.7f        %10.7f\n",z[i+2],z[i+3]);
  }
analytic(x,y,n);
printf("\n Real Part of Analytic Signal\n");
for (i=0;i<n/2;i+=4)
  { printf("        %10.7f        %10.7f",x[i],x[i+1]);
    printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
  }
printf("\n Imaginary Part of Analytic Signal\n");
for (i=0;i<n/2;i+=4)
  { printf("        %10.7f        %10.7f",y[i],y[i+1]);
    printf("        %10.7f        %10.7f\n",y[i+2],y[i+3]);
  }
}
```

运行结果：

原始信号 $x(i)$(前 32 个数据)

0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847
1.0000000	0.9951847	0.9807853	0.9569403
0.9238795	0.8819213	0.8314696	0.7730105
0.7071068	0.6343933	0.5555702	0.4713967
0.3826834	0.2902847	0.1950903	0.0980171

序列 $x(i)$ 的理想希尔伯特变换(前 32 个数据)

−1.0000000	−0.9951847	−0.9807853	−0.9569403
−0.9238795	−0.8819213	−0.8314696	−0.7730105
−0.7071068	−0.6343933	−0.5555702	−0.4713967
−0.3826834	−0.2902847	−0.1950903	−0.0980171
−0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847

解析信号的实部(前 32 个数据)

−0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847
1.000000	0.9951847	0.9807853	0.9569403
0.9238795	0.8819213	0.8314696	0.7730105
0.7071068	0.6343933	0.5555702	0.4713967
0.3826834	0.2902847	0.1950903	0.0980171

解析信号的虚部(前 32 个数据)

−1.0000000	−0.9951847	−0.9807853	−0.9569403
−0.9238795	−0.8819213	−0.8314696	−0.7730105
−0.7071068	−0.6343933	−0.5555702	−0.4713967
−0.3826834	−0.2902847	−0.1950903	−0.0980171
−0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847

第二篇　数字信号处理

第一章　快速傅立叶变换

§1.1　离散傅立叶变换

一、功　能

计算复序列的离散傅立叶变换(DFT)和离散傅立叶反变换(IDFT)

二、方法简介

序列 $x(n)(n=0,1,\cdots,N-1)$的离散傅立叶变换定义为

$$X(k)=\sum_{n=0}^{N-1}x(n)\,e^{-j\frac{2\pi nk}{N}}$$

设 $x(n)=a(n)+j\,b(n)$, $X(k)=A(k)+j\,B(k)$, $Q=2\pi/N$，则上式变为

$$A(k)+j\,B(k)=\sum_{n=0}^{N-1}[a(n)+j\,b(n)][\cos(Qnk)-j\sin(Qnk)]$$

即

$$A(k)=\sum_{n=0}^{N-1}[a(n)\cos(Qnk)+b(n)\sin(Qnk)]$$

$$B(k)=\sum_{n=0}^{N-1}[b(n)\cos(Qnk)-a(n)\sin(Qnk)]$$

序列 $X(k)$的离散傅立叶反变换定义为

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)W_N^{-nk},\quad n=0,1,\cdots,N-1$$

它与离散傅立叶正变换的区别在于将 W_N改变为 W_N^{-1}，并多了一个除以 N 的运算。计算公式如下：

$$a(n)=\frac{1}{N}\sum_{k=0}^{N-1}[A(k)\cos(Qnk)-B(k)\sin(Qnk)]$$

$$b(n)=\frac{1}{N}\sum_{k=0}^{N-1}[B(k)\cos(Qnk)+A(k)\sin(Qnk)].$$

三、使用说明

1. 子函数语句

```
void dft(x,y,a,b,n,sign)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。存放要变换数据的实部。

y ——双精度实型一维数组，长度为 n。存放要变换数据的虚部。

a ——双精度实型一维数组，长度为 n。存放变换结果的实部。

b ——双精度实型一维数组，长度为 n。存放变换结果的虚部。

n ——整型变量。数据长度。

sign——整型变量。当 $sign=1$ 时，子函数 *dft*()计算离散傅立叶正变换；当 $sign=-1$时，子函数 *dft*()计算离散傅立叶反变换。

四、子函数程序(文件名:dft.c)

```
#include "math.h"
void dft(x,y,a,b,n,sign)
int n,sign;
double x[],y[],a[],b[];
{ int i,k;
  double c,d,q,w,s;
  q = 6.28318530718/n;
  for (k=0;k<n;k++)
    { w = k * q;
      a[k] = b[k] = 0.0;
      for (i=0;i<n;i++)
        { d = i * w;
          c = cos(d);
          s = sin(d) * sign;
          a[k] += c * x[i] + s * y[i];
          b[k] += c * y[i] - s * x[i];
        }
    }
  if ( sign == -1 )
    { c = 1.0/n;
      for (k=0;k<n;k++)
        { a[k] = c * a[k];
```

```
            b[k] = c * b[k];
          }
      }
  }
```

五、例　题

设输入序列 $x(i)$ 为 $x(i)=Q^i$ ，$i=0,1,\cdots,n-1$

其离散傅立叶变换为 $X(k)=\frac{1-Q^n}{1-QW^k}$ ，$k=0,1,\cdots,n-1$

这里 $W=e^{-j\frac{2\pi}{n}}$。选取 $Q=0.9+j\,0.3$，$n=32$，计算 $x(i)$ 的离散傅立叶变换 $X(k)$ 和 $X(k)$ 的离散傅立叶反变换 $x(i)$，并与理论值进行比较。

主函数程序(文件名:dft.m):

```
#include "math.h"
#include "dft.c"
main()
{ int i,j,n;
  double a1,a2,c,c1,c2,d1,d2,q1,q2,w,w1,w2;
  double x[32],y[32],a[32],b[32];
  n = 32;
  a1 = 0.9;
  a2 = 0.3;
  x[0] = 1.0;
  y[0] = 0.0;
  for (i=1;i<n;i++)
    { x[i] = a1 * x[i-1] - a2 * y[i-1];
      y[i] = a2 * x[i-1] + a1 * y[i-1];
    }
  printf("\n Original Sequence\n");
  for (i=0;i<n/2;i++)
    { for (j=0;j<2;j++)
        printf("      %10.7f  + J  %10.7f",x[2 * i+j],y[2 * i+j]);
      printf("\n");
    }
  q1 = x[n-1];
  q2 = y[n-1];
  dft(x,y,a,b,n,1);
  printf("\n Discrete Fourier Transform\n");
  for (i=0;i<n/2;i++)
```

```
    { for (j=0;j<2;j++)
        printf("        %10.7f   + J   %10.7f",a[2*i+j],b[2*i+j]);
      printf("\n");
    }
  for (i=0;i<n;i++)
    { w = 6.28318530718/n*i;
      w1 = cos(w);
      w2 =-sin(w);
      c1 = 1.0 - a1*w1 + a2*w2;
      c2 = a1*w2 + a2*w1;
      c = c1*c1 + c2*c2;
      d1 = 1.0-a1*q1 + a2*q2;
      d2 - a1*q2 + a2*q1;
      x[i] = (c1*d1 + c2*d2)/c;
      y[i] = (c2*d1 - c1*d2)/c;
    }
  printf("\n Theoretical Discrete Fourier Transform\n");
  for (i=0;i<n/2;i++)
    { for (j=0;j<2;j++)
        printf("        %10.7f   + J   %10.7f",x[2*i+j],y[2*i+j]);
      printf("\n");
    }
  dft(a,b,x,y,n,-1);
  printf("\n Inverse Discrete Fourier Transform\n");
  for (i=0;i<n/2;i++)
    { for (j=0;j<2;j++)
        printf("        %10.7f   + J   %10.7f",x[2*i+j],y[2*i+j]);
      printf("\n");
    }
}
```

运行结果：

输入序列 $x(i)$ $(i=0,1,\cdots,31)$

```
 1.0000000  + J   0.0000000     0.9000000  + J   0.3000000
 0.7200000  + J   0.5400000     0.4860000  + J   0.7020000
 0.2268000  + J   0.7776000    -0.0291600  + J   0.7678800
-0.2566080  + J   0.6823440    -0.4356504  + J   0.5371272
-0.5532235  + J   0.3527194    -0.6037170  + J   0.1514804
-0.5887894  + J  -0.0447828    -0.5164756  + J  -0.2169413
```

−0.3997457	+J	−0.3501899	−0.2547141	+J	−0.4350946
−0.0987144	+J	−0.4679994	0.0515569	+J	−0.4508137
0.1816453	+J	−0.3902653	0.2805604	+J	−0.2967452
0.3415279	+J	−0.1829025	0.3622459	+J	−0.0621539
0.3446674	+J	0.0527352	0.2943801	+J	0.1508619
0.2196835	+J	0.2240898	0.1304882	+J	0.2675859
0.0371637	+J	0.2799738	−0.0505448	+J	0.2631255
−0.1244280	+J	0.2216495	−0.1784800	+J	0.1621561
−0.2092789	+J	0.0923965	−0.2160699	+J	0.0203732
−0.2005749	+J	−0.0464851	−0.1665719	+J	−0.1020091

$x(i)$的离散傅立叶变换$X(k)(k=0,1,\cdots,31)$为

0.6939728	+J	3.4997157	2.7922679	+J	8.0504557
9.4029646	+J	−9.1350136	1.8664455	+J	−3.8338328
1.1318227	+J	−2.2341573	0.9047939	+J	−1.5346293
0.7995572	+J	−1.1396094	0.7396056	+J	−0.8823144
0.7008616	+J	−0.6985654	0.6735758	+J	−0.5584785
0.6531094	+J	−0.4462450	0.6369913	+J	−0.3526891
0.6237884	+J	−0.2720860	0.6126129	+J	−0.2006419
0.6028833	+J	−0.1357032	0.5942004	+J	−0.0753137
0.5862775	+J	−0.0179492	0.5788996	+J	0.0376517
0.5718985	+J	0.0926070	0.5651358	+J	0.1479833
0.5584921	+J	0.2048808	0.5518591	+J	0.2645222
0.5451336	+J	0.3283649	0.5382144	+J	0.3982571
0.5310015	+J	0.4766778	0.5234037	+J	0.5671323
0.5153625	+J	0.6748500	0.5069258	+J	0.8081015
0.4984670	+J	0.9809063	0.4913894	+J	1.2192074
0.4907322	+J	1.5770820	0.5173540	+J	2.1888329

$x(i)$的离散傅立叶变换$X(k)(k=0,1,\cdots,31)$的理论值为

0.6939728	+J	3.4997157	2.7922679	+J	8.0504557
9.4029646	+J	−9.1350136	1.8664455	+J	−3.8338328
1.1318227	+J	−2.2341573	0.9047939	+J	−1.5346293
0.7995572	+J	−1.1396094	0.7396056	+J	−0.8823144
0.7008616	+J	−0.6985654	0.6735758	+J	−0.5584785
0.6531094	+J	−0.4462450	0.6369913	+J	−0.3526891
0.6237884	+J	−0.2720860	0.6126129	+J	−0.2006419
0.6028833	+J	−0.1357032	0.5942004	+J	−0.0753137
0.5862775	+J	−0.0179492	0.5788996	+J	0.0376517
0.5718985	+J	0.0926070	0.5651358	+J	0.1479833

0.5584921	+ J	0.2048808	0.5518591	+ J	0.2645222
0.5451336	+ J	0.3283649	0.5382144	+ J	0.3982571
0.5310015	+ J	0.4766778	0.5234037	+ J	0.5671323
0.5153625	+ J	0.6748500	0.5069258	+ J	0.8081015
0.4984670	+ J	0.9809063	0.4913894	+ J	1.2192074
0.4907322	+ J	1.5770820	0.5173540	+ J	2.1888329

$X(k)$的离散傅立叶反变换 $x(i)$ $(i=0,1,\cdots,31)$为

1.0000000	+ J	0.0000000	0.9000000	+ J	0.3000000
0.7200000	+ J	0.5400000	0.4860000	+ J	0.7020000
0.2268000	+ J	0.7776000	−0.0291600	+ J	0.7678800
−0.2566080	+ J	0.6823440	−0.4356504	+ J	0.5371272
−0.5532235	+ J	0.3527194	−0.6037170	+ J	0.1514804
−0.5887894	+ J	−0.0447828	−0.5164756	+ J	−0.2169413
−0.3997457	+ J	−0.3501899	−0.2547141	+ J	−0.4350946
−0.0987144	+ J	−0.4679994	0.0515569	+ J	−0.4508137
0.1816453	+ J	−0.3902653	0.2805604	+ J	−0.2967452
0.3415279	+ J	−0.1829025	0.3622459	+ J	−0.0621539
0.3446674	+ J	0.0527352	0.2943801	+ J	0.1508619
0.2196835	+ J	0.2240898	0.1304882	+ J	0.2675859
0.0371637	+ J	0.2799738	−0.0505448	+ J	0.2631255
−0.1244280	+ J	0.2216495	−0.1784800	+ J	0.1621561
−0.2092789	+ J	0.0923965	−0.2160699	+ J	0.0203732
−0.2005749	+ J	−0.0464851	−0.1665719	+ J	−0.1020091

§1.2 快速傅立叶变换

一、功　能

计算复序列的快速傅立叶变换

二、方法简介

序列 $x(n)$的离散傅立叶变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},\quad k=0,1,\cdots,N-1$$

将序列 $x(n)$按序号 n 的奇偶分成两组，即

$$\left.\begin{aligned}x_1(n)&=x(2n)\\x_2(n)&=x(2n+1)\end{aligned}\right\}\quad n=0,1,\cdots,\frac{N}{2}-1$$

因此，$x(n)$的傅立叶变换可写成为

$$X(k)=\sum_{\substack{n=0\\ n\text{为偶数}}}^{N-1}x(n)W_N^{nk}+\sum_{\substack{n=0\\ n\text{为奇数}}}^{N-1}x(n)W_N^{nk}$$

$$=\sum_{n=0}^{N/2-1}x(2n)W_N^{2nk}+\sum_{n=0}^{N/2-1}x(2n+1)W_N^{(2n+1)k}$$

$$=\sum_{n=0}^{N/2-1}x_1(n)W_{N/2}^{nk}+W_N^k\sum_{n=0}^{N/2-1}x_2(n)W_{N/2}^{nk}$$

由此可得 $X(k)=X_1(k)+W_N^kX_2(k)$ ， $k=0,1,\cdots,\frac{N}{2}-1$

式中

$$X_1(k)=\sum_{n=0}^{N/2-1}x(2n)W_{N/2}^{nk},$$

$$X_2(k)=\sum_{n=0}^{N/2-1}x(2n+1)W_{N/2}^{nk}$$

它们分别是 $x_1(n)$和 $x_2(n)$的 $N/2$ 点 DFT。上面的推导表明：一个 N 点的 DFT 被分解为两个 $N/2$ 点的 DFT，这两个 $N/2$ 点的 DFT 又可合成为一个 N 点的 DFT。但上面给出的公式仅能得到 $X(k)$的前 N/2 点的值，要用 $X_1(k)$和 $X_2(k)$来表达$X(k)$的后半部分的值，还必须运用权系数 W_N 的周期性与对称性，即

$$W_{N/2}^{n(k+N/2)}=W_{N/2}^{nk},W_N^{(k+N/2)}=-W_N^k$$

因此，$X(k)$的后 $N/2$ 点的值可表示为

$$X\left(k+\frac{N}{2}\right)=X_1\left(k+\frac{N}{2}\right)+W_N^{k+\frac{N}{2}}X_2\left(k+\frac{N}{2}\right)$$

$$=X_1(k)-W_N^kX_2(k)\quad,\quad k=0,1,\cdots,\frac{N}{2}-1$$

通过上面的推导可以看出，N 点的 DFT 可以分解为两个 $N/2$ 点的 DFT，每个 $N/2$ 点的 DFT 又可以分解为两个 $N/4$ 点的 DFT。依此类推，当 N 为 2 的整数次幂时（$N=2^M$），由于每分解一次降低一阶幂次，所以通过 M 次的分解，最后全部成为一系列 2 点 DFT 运算。以上就是按时间抽取的快速傅立叶变换(FFT)算法。

序列 $X(k)$的离散傅立叶反变换为

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)W_N^{-nk},\quad n=0,1,\cdots,N-1$$

它与离散傅立叶正变换的区别在于将 W_N改变为 W_N^{-1}，并多了一个除以 N 的运算。因为 W_N和 W_N^{-1}对于推导按时间抽取的快速傅立叶变换算法并无实质性区别，因此可将 FFT 和快速傅立叶反变换(IFFT)算法合并在同一个程序中。

三、使用说明

1. 子函数语句

```
void fft(x,y,n,sign)
```

2. 形参说明

x ——双精度实型一维数组，长度为n。开始时存放要变换数据的实部，最后存放变换结果的实部。

y ——双精度实型一维数组，长度为n。开始时存放要变换数据的虚部，最后存放变换结果的虚部。

n ——整型变量。数据长度，必须是2的整数次幂，即$n=2^m$。

sign ——整型变量。当sign=1时，子函数fft()计算离散傅立叶正变换（DFT）；当sign=−1时，子函数fft()计算离散傅立叶反变换（IDFT）。

四、子函数程序(文件名:fft.c)

```
#include "math.h"
void fft(x,y,n,sign)
int n,sign;
double x[],y[];
{ int i,j,k,l,m,n1,n2;
  double c,c1,e,s,s1,t,tr,ti;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2*j;
      if (j==n) break;
    }
  n1 = n-1;
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { tr = x[j];
          ti = y[j];
          x[j] = x[i];
          y[j] = y[i];
          x[i] = tr;
          y[i] = ti;
        }
      k = n/2;
      while ( k < (j+1) )
        { j = j-k;
          k = k/2;
        }
      j = j+k;
```

```
    }

  n1 = 1;
  for (l=1;l<=m;l++)
    { n1 = 2 * n1;
      n2 = n1/2;
      e = 3.14159265359/n2;
      c = 1.0;
      s = 0.0;
      c1 = cos(e);
      s1 =-sign * sin(e);
      for (j=0;j<n2;j++)
        {for (i=j;i<n;i+=n1)
            { k = i + n2;
              tr = c * x[k] - s * y[k];
              ti = c * y[k] + s * x[k];
              x[k] = x[i] - tr;
              y[k] = y[i] - ti;
              x[i] = x[i] + tr;
              y[i] = y[i] + ti;
            }
          t = c;
          c = c * c1 - s * s1;
          s = t * s1 + s * c1;
        }
    }
  if ( sign == -1 )
    { for (i=0;i<n;i++)
        { x[i] /= n;
          y[i] /= n;
        }
    }
}
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i) = Q^i \quad , \quad i = 0,1,\cdots,n-1$$

其离散傅立叶变换为

$$X(k) = \frac{1 - Q^n}{1 - QW^k} \quad , \quad k = 0,1,\cdots,n-1$$

这里 $W = e^{-j\frac{2\pi}{n}}$。选取 $Q=0.9+j\,0.3$，$n=32$，计算离散傅立叶变换 $X(k)$和离散傅立叶反变换 $x(i)$，并和理论值进行比较。

主函数程序(文件名:fft.m):

```
#include "math.h"
#include "fft.c"
main()
{ int i,j,n;
  double a1,a2,c,c1,c2,d1,d2,q1,q2,w,w1,w2;
  double x[32],y[32],a[32],b[32];
  n = 32;
  a1 = 0.9;
  a2 = 0.3;
  x[0] = 1.0;
  y[0] = 0.0;
  for (i=1;i<n;i++)
    { x[i] = a1 * x[i-1] - a2 * y[i-1];
      y[i] = a2 * x[i-1] + a1 * y[i-1];
    }
  printf("\n COMPLEX INPUT SEQUENCE\n");
  for (i=0;i<n/2;i++)
    { for (j=0;j<2;j++)
          printf(" %10.7f + J %10.7f",x[2 * i+j],y[2 * i+j]);
      printf("\n");
    }
  q1 = x[n-1];
  q2 = y[n-1];
  for (i=0;i<n;i++)
    { w = 6.28318530718/n * i;
      w1 = cos(w);
      w2 =-sin(w);
      c1 = 1.0 - a1 * w1 + a2 * w2;
      c2 = a1 * w2 + a2 * w1;
      c = c1 * c1 + c2 * c2;
      d1 = 1.0-a1 * q1 + a2 * q2;
      d2 = a1 * q2 + a2 * q1;
      a[i] = (c1 * d1 + c2 * d2)/c;
```

```
        b[i] = (c2 * d1 - c1 * d2)/c;
      }
    printf("\n THEORETICAL DFT\n");
    for (i=0;i<n/2;i++)
      { for (j=0;j<2;j++)
          printf(" %10.7f + J %10.7f",a[2 * i+j],b[2 * i+j]);
        printf("\n");
      }
    fft(x,y,n,1);
    printf("\n DISCRETE FOURIER TRANSFORM\n");
    for (i=0;i<n/2;i++)
      { for (j=0;j<2;j++)
          printf(" %10.7f + J %10.7f",x[2 * i+j],y[2 * i+j]);
        printf("\n");
      }
  fft(x,y,n,-1);
    printf("\n INVERSE DISCRETE FOURIER TRANSFORM\n");
    for (i=0;i<n/2;i++)
      { for (j=0;j<2;j++)
          printf(" %10.7f + J %10.7f",x[2 * i+j],y[2 * i+j]);
        printf("\n");
      }
  }
```

运行结果为：

输入序列 $x(i)$ 为

```
  1.0000000  + J   0.0000000     0.9000000  + J   0.3000000
  0.7200000  + J   0.5400000     0.4860000  + J   0.7020000
  0.2268000  + J   0.7776000    -0.0291600  + J   0.7678800
 -0.2566080  + J   0.6823440    -0.4356504  + J   0.5371272
 -0.5532235  + J   0.3527194    -0.6037170  + J   0.1514804
 -0.5887894  + J  -0.0447828    -0.5164756  + J  -0.2169413
 -0.3997457  + J  -0.3501899    -0.2547141  + J  -0.4350946
 -0.0987144  + J  -0.4679994     0.0515569  + J  -0.4508137
  0.1816453  + J  -0.3902653     0.2805604  + J  -0.2967452
  0.3415279  + J  -0.1829025     0.3622459  + J  -0.0621539
  0.3446674  + J   0.0527352     0.2943801  + J   0.1508619
  0.2196835  + J   0.2240898     0.1304882  + J   0.2675859
  0.0371637  + J   0.2799738    -0.0505448  + J   0.2631255
```

−0.1244280	+J	0.2216495	−0.1784800	+J	0.1621561
−0.2092789	+J	0.0923965	−0.2160699	+J	0.0203732
−0.2005749	+J	−0.0464851	−0.1665719	+J	−0.1020091

$x(i)$的离散傅立叶变换$X(k)(k=0,1,\cdots,31)$的理论值为

0.6939728	+J	3.4997157	2.7922679	+J	8.0504557
9.4029646	+J	−9.1350136	1.8664455	+J	−3.8338328
1.1318227	+J	−2.2341573	0.9047939	+J	−1.5346293
0.7995572	+J	−1.1396094	0.7396056	+J	−0.8823144
0.7008616	+J	−0.6985654	0.6735758	+J	−0.5584785
0.6531094	+J	−0.4462450	0.6369913	+J	−0.3526891
0.6237884	+J	−0.2720860	0.6126129	+J	−0.2006419
0.6028833	+J	−0.1357032	0.5942004	+J	−0.0753137
0.5862775	+J	−0.0179492	0.5788996	+J	0.0376517
0.5718985	+J	0.0926070	0.5651358	+J	0.1479833
0.5584921	+J	0.2048808	0.5518591	+J	0.2645222
0.5451336	+J	0.3283649	0.5382144	+J	0.3982571
0.5310015	+J	0.4766778	0.5234037	+J	0.5671323
0.5153625	+J	0.6748500	0.5069258	+J	0.8081015
0.4984670	+J	0.9809063	0.4913894	+J	1.2192074
0.4907322	+J	1.5770820	0.5173540	+J	2.1888329

离散傅立叶变换$X(k)(k=0,1,\cdots,31)$为

0.6939728	+J	3.4997157	2.7922679	+J	8.0504557
9.4029646	+J	−9.1350136	1.8664455	+J	−3.8338328
1.1318227	+J	−2.2341573	0.9047939	+J	−1.5346293
0.7995572	+J	−1.1396094	0.7396056	+J	−0.8823144
0.7008616	+J	−0.6985654	0.6735758	+J	−0.5584785
0.6531094	+J	−0.4462450	0.6369913	+J	−0.3526891
0.6237884	+J	−0.2720860	0.6126129	+J	−0.2006419
0.6028833	+J	−0.1357032	0.5942004	+J	−0.0753137
0.5862775	+J	−0.0179492	0.5788996	+J	0.0376517
0.5718985	+J	0.0926070	0.5651358	+J	0.1479833
0.5584921	+J	0.2048808	0.5518591	+J	0.2645222
0.5451336	+J	0.3283649	0.5382144	+J	0.3982571
0.5310015	+J	0.4766778	0.5234037	+J	0.5671323
0.5153625	+J	0.6748500	0.5069258	+J	0.8081015
0.4984670	+J	0.9809063	0.4913894	+J	1.2192074

0.4907322	+J	1.5770820	0.5173540	+J	2.1888329

离散傅立叶反变换 $x(i)(i=0,1,\cdots,31)$为

1.0000000	+J	−0.0000000	0.9000000	+J	0.3000000
0.7200000	+J	0.5400000	0.4860000	+J	0.7020000
0.2268000	+J	0.7776000	−0.0291600	+J	0.7678800
−0.2566080	+J	0.6823440	−0.4356504	+J	0.5371272
−0.5532235	+J	0.3527194	−0.6037170	+J	0.1514804
−0.5887894	+J	−0.0447828	−0.5164756	−J	−0.2169413
−0.3997457	+J	−0.3501899	−0.2547141	+J	−0.4350946
−0.0987144	+J	−0.4679994	0.0515569	+J	−0.4508137
0.1816453	+J	−0.3902653	0.2805604	+J	−0.2967452
0.3415279	+J	−0.1829025	0.3622459	+J	−0.0621539
0.3446674	+J	0.0527352	0.2943801	+J	0.1508619
0.2196835	+J	0.2240898	0.1304882	+J	0.2675859
0.0371637	+J	0.2799738	−0.0505448	+J	0.2631255
−0.1244280	+J	0.2216495	−0.1784800	+J	0.1621561
−0.2092789	+J	0.0923965	−0.2160699	+J	0.0203732
−0.2005749	+J	−0.0464851	−0.1665719	+J	−0.1020091

§1.3 基4快速傅立叶变换

一、功 能

计算复序列的基4快速傅立叶变换

二、方法简介

序列 $x(n)$的离散傅立叶变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},\quad k=0,1,\cdots,N-1$$

若 $N=4^M$,则将序列 $x(n)$分成四个 $N/4$ 点的序列 $x_1(n)$、$x_2(n)$、$x_3(N)$、$x_4(i)$($n=0,1,\cdots,N/4-1$),即

$$x(n)=x_1(n)+x_2(n)+x_3(n)+x_4(n)$$

式中

$$\begin{cases} x_1(n) = x(n) & , \quad (n = 0,1,\cdots,N/4-1) \\ x_2(n) = x(n+\dfrac{N}{4}) & , \quad (n = 0,1,\cdots,N/4-1) \\ x_3(n) = x(n+\dfrac{N}{2}) & , \quad (n = 0,1,\cdots,N/4-1) \\ x_4(n) = x(n+\dfrac{3N}{4}) & , \quad (n = 0,1,\cdots,N/4-1) \end{cases}$$

把 $x(n)$代入 DFT 的表达式中，则有

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/4-1}[x_1(n)W_N^{nk} + x_2(n)W_N^{(n+N/4)k} + x_3(n)W_N^{(n-N/2)k} + x_4(n)W_N^{(n+3N/4)k}] \\ &= \sum_{n=0}^{N/4-1}[x_1(n) + x_2(n)W_N^{Nk/4} + x_3(n)W_N^{Nk/2} + x_4(n)W_N^{3Nk/4}]W_N^{nk} \end{aligned}$$

$$(k = 0,1,\cdots,N-1)$$

把 $X(k)$按频率抽取，得

$$\begin{cases} X(4k) = \displaystyle\sum_{n=0}^{N/4-1}[x_1(n) + x_2(n) + x_3(n) + x_4(n)]W_{N/4}^{nk} \\ X(4k+1) = \displaystyle\sum_{n=0}^{N/4-1}[x_1(n) - j\,x_2(n) - x_3(n) + j\,x_4(n)]W_N^{n}W_{N/4}^{nk} \\ X(4k+2) = \displaystyle\sum_{n=0}^{N/4-1}[x_1(n) - x_2(n) + x_3(n) - x_4(n)]W_N^{2n}W_{N/4}^{nk} \\ X(4k+3) = \displaystyle\sum_{n=0}^{N/4-1}[x_1(n) + j\,x_2(n) - x_3(n) - j\,x_4(n)]W_N^{3n}W_{N/4}^{nk} \end{cases}$$

$$(k = 0,1,\cdots,\frac{N}{4}-1)$$

通过上面的分解，可求得所有 $X(k)$值，其基本运算式为

$$\begin{cases} f_1(n) = x_1(n) + x_2(n) + x_3(n) + x_4(n) \\ f_2(n) = [x_1(n) - j\,x_2(n) - x_3(n) + j\,x_4(n)]W_N^{n} \\ f_3(n) = [x_1(n) - x_2(n) + x_3(n) - x_4(n)]W_N^{2n} \\ f_4(n) = [x_1(n) + j\,x_2(n) - x_3(n) - j\,x_4(n)]W_N^{3n} \end{cases}$$

$$(k = 0,1,\cdots,\frac{N}{4}-1)$$

这样，就将一个 N 点的 DFT 转化为四个 $N/4$ 点 DFT 来计算。依此类推，直至分解到最后一级。以上就是按频率抽取的基 4 快速傅立叶变换算法。与基 2 FFT 相比，基 4 FFT 的乘法量约减少 25%，加法量也略有减少。

三、使用说明

1. 子函数语句

```
void fft4(x,y,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换数据的实部，最后存放变换结果的实部。

y ——双精度实型一维数组，长度为 n。开始时存放要变换数据的虚部，最后存放变换结果的虚部。

n ——整型变量。数据长度，必须是 4 的整数次幂，即 $n=4^m$。

四、子函数程序(文件名:fft4.c)

```
#include "math.h"
void fft4(x,y,n)
int n;
double x[],y[];
{ int i,j,k,m,i1,i2,i3,n1,n2;
  double a,b,c,e,r1,r2,r3,r4,s1,s2,s3,s4;
  double co1,co2,co3,si1,si2,si3;
  for (j=1,i=1;i<10;i++)
    { m = i;
      j = 4 * j;
      if (j==n) break;
    }
  n2 = n;
  for (k=1;k<=m;k++)
    { n1 = n2;
      n2 = n2/4;
      e = 6.28318530718/n1;
      a = 0;
      for (j=0;j<n2;j++)
        { b = a + a;
          c = a + b;
          co1 = cos(a);
          co2 = cos(b);
          co3 = cos(c);
          si1 = sin(a);
          si2 = sin(b);
          si3 = sin(c);
          a = (j+1) * e;
          for (i=j;i<n;i=i+n1)
```

```
            { i1 = i + n2;
              i2 = i1 + n2;
              i3 = i2 + n2;
              r1 = x[i] + x[i2];
              r3 = x[i] - x[i2];
              s1 = y[i] + y[i2];
              s3 = y[i] - y[i2];
              r2 = x[i1] + x[i3];
              r4 = x[i1] - x[i3];
              s2 = y[i1] + y[i3];
              s4 = y[i1] - y[i3];
              x[i] = r1 - r2;
              r2 - r1 - r2;
              r1 = r3 - s4;
              r3 = r3 + s4;
              y[i] = s1 + s2;
              s2 = s1 - s2;
              s1 = s3 + r4;
              s3 = s3 - r4;
              x[i1] = co1 * r3 + si1 * s3;
              y[i1] = co1 * s3 - si1 * r3;
              x[i2] = co2 * r2 + si2 * s2;
              y[i2] = co2 * s2 - si2 * r2;
              x[i3] = co3 * r1 + si3 * s1;
              y[i3] = co3 * s1 - si3 * r1;
            }
        }
    }
  n1 = n-1;
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { r1 = x[j];
          s1 = y[j];
          x[j] = x[i];
          y[j] = y[i];
          x[i] = r1;
          y[i] = s1;
        }
```

```
        k = n/4;
        while ( 3 * k < (j+1) )
          { j = j - 3 * k;
            k = k/4;
          }
        j = j + k;
      }
  }
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i) = Q^i \quad , \quad i = 0,1,\cdots,n-1$$

其离散傅立叶变换为

$$X(k) = \frac{1-Q^n}{1-QW^k} \quad , \quad k = 0,1,\cdots,n-1$$

这里 $W = e^{-j\frac{2\pi}{n}}$。选取 $Q=0.9+j\,0.3$，$n=64$，用基 4 快速算法计算离散傅立叶变换 $X(k)$。

主函数程序(文件名:fft4.m):

```
  #include "math.h"
  #include "fft4.c"
  main()
  { int i,j,n;
    double a1,a2,x[64],y[64];
    n = 64;
    a1 = 0.9;
    a2 = 0.3;
    x[0] = 1.0;
    y[0] = 0.0;
    for (i=1;i<n;i++)
      { x[i] = a1 * x[i-1] - a2 * y[i-1];
        y[i] = a2 * x[i-1] + a1 * y[i-1];
      }
    fft4(x,y,n);
    printf("\n DISCRETE FOURIER TRANSFORM\n");
    for (i=0;i<n/2;i++)
      { for (j=0;j<2;j+-)
            printf(" %10.7f + J %10.7f",x[2*i+j],y[2*i+j]);
          printf("\n");
      }
```

```
}
```

运行结果：

离散傅立叶变换 $X(k)(k=0,1,\cdots,63)$ 为

```
1.1073623   + J    2.9837664        1.6544123   + J    4.1927516
3.6005083   + J    6.6940530       15.8380686   + J    7.2617105
6.9859186   + J   -9.3782468        2.0142267   + J   -5.4811879
1.1001964   + J   -3.6410347        0.8063639   + J   -2.6956512
0.6800245   + J   -2.1280651        0.6157312   + J   -1.7500873
0.5792619   + J   -1.4798111        0.5569682   + J   -1.2763197
0.5425870   + J   -1.1170013        0.5329358   + J   -0.9883727
0.5262675   + J   -0.8819048        0.5215631   + J   -0.7919419
0.5181981   + J   -0.7145861        0.5157743   + J   -0.6470634
0.5140292   + J   -0.5873447        0.5127847   + J   -0.5339108
0.5119172   + J   -0.4856003        0.5113388   + J   -0.4415091
0.5109864   + J   -0.4009215        0.5108138   + J   -0.3632625
0.5107867   + J   -0.3280634        0.5108797   + J   -0.2949369
0.5110739   + J   -0.2635590        0.5113549   + J   -0.2336549
0.5117122   + J   -0.2049888        0.5121381   + J   -0.1773554
0.5126273   + J   -0.1505734        0.5131761   + J   -0.1244808
0.5137827   + J   -0.0989299        0.5144465   + J   -0.0737846
0.5151682   + J   -0.0489168        0.5159496   + J   -0.0242037
0.5167939   + J    0.0004742        0.5177053   + J    0.0252361
0.5186893   + J    0.0502023        0.5197531   + J    0.0754977
0.5209053   + J    0.1012532        0.5221565   + J    0.1276093
0.5235196   + J    0.1547191        0.5250104   + J    0.1827519
0.5266482   + J    0.2118982        0.5284563   + J    0.2423748
0.5304637   + J    0.2744324        0.5327066   + J    0.3083645
0.5352299   + J    0.3445192        0.5380910   + J    0.3833143
0.5413633   + J    0.4252586        0.5451421   + J    0.4709800
0.5495536   + J    0.5212643        0.5547674   + J    0.5771111
0.5610159   + J    0.6398135        0.5686255   + J    0.7110769
0.5780666   + J    0.7931999        0.5900385   + J    0.8893589
0.6056196   + J    1.0040723        0.6265482   + J    1.1439945
0.6557802   + J    1.3193413        0.6986936   + J    1.5466251
0.7659596   + J    1.8543286        0.8813627   + J    2.2959168
```

§1.4 分裂基快速傅立叶变换

一、功 能

计算复序列的分裂基快速傅立叶变换

二、方法简介

序列 $x(n)$的离散傅立叶变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},\quad k=0,1,\cdots,N-1$$

将 $X(k)$按序号 k 的奇偶分成两组。当 k 为偶数时，进行基 2 频率抽取分解，$X(k)$可表示为

$$X(2k)=\sum_{n=0}^{N/2-1}[x(n)+x(n+\frac{N}{2})]W_N^{2nk}\quad,\quad k=0,1,\cdots,\frac{N}{2}-1$$

当 k 为奇数时，进行基 4 频率抽取分解，$X(k)$可表示为

$$\begin{cases}X(4k+1)=\sum_{n=0}^{N/4-1}\{[x(n)-x(n+\frac{N}{2})]-j\,[x(n+\frac{N}{4})-x(n+\frac{3N}{4})]\}W_N^{n}W_N^{4nk}\\[2ex] X(4k+3)=\sum_{n=0}^{N/4-1}\{[x(n)-x(n+\frac{N}{2})]+j\,[x(n+\frac{N}{4})-x(n+\frac{3N}{4})]\}W_N^{3n}W_N^{4nk}\end{cases}$$

$$k=0,1,\cdots,\frac{N}{4}-1$$

由此可见，一个 N 点的 DFT 可以分解为一个 N/2 点的 DFT 和两个 $N/4$ 点的 DFT。这种分解既有基 2 的部分，又有基 4 的部分，因此称为分裂基分解。上面的 $N/2$ 点 DFT 又可分解为一个 $N/4$ 点的 DFT 和两个 $N/8$ 点的 DFT，而两个 $N/4$ 点的 DFT 也分别可以分解为一个 $N/8$ 点的 DFT 和两个 $N/16$ 点的 DFT。依此类推，直至分解到最后一级为止。这就是按频率抽取的分裂基快速傅立叶变换算法。。

分裂基快速算法是将基 2 和基 4 分解组合而成。在基 2^m 类快速算法中，分裂基算法具有最少的运算量，且仍保留结构规则、原位计算等优点。

三、使用说明

1. 子函数语句

```
void srfft(x,y,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换数据的实部，最后存放变换结果的实部。

y ——双精度实型一维数组，长度为 n。开始时存放要变换数据的虚部，最后存放

变换结果的虚部。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:srfft.c)

```
#include "math.h"
void srfft(x,y,n)
int n;
double x[],y[];
{ int i,j,k,m,i1,i2,i3,n1,n2,n4,id,is;
  double a,b,c,e,a3,r1,r2,r3,r4;
  double c1,c3,s1,s2,s3,cc1,cc3,ss1,ss3;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2 * j;
      if (j==n) break;
    }
  n2 = 2 * n;
  for (k=1;k<m;k++)
    { n2 = n2/2;
      n4 = n2/4;
      e = 6.28318530718/n2;
      a = 0;
      for (j=0;j<n4;j++)
        { a3 = 3 * a;
          cc1 = cos(a);
          ss1 = sin(a);
          cc3 = cos(a3);
          ss3 = sin(a3);
          a = (j-1) * e;
          is = j;
          id = 2 * n2;
          do {
               for (i=is;i<(n-1);i=i+id)
                 { i1 = i + n4;
                   i2 = i1 + n4;
                   i3 = i2 + n4;
                   r1 = x[i] - x[i2];
                   x[i] = x[i] + x[i2];
```

```
                r2 = x[i1] - x[i3];
                x[i1] = x[i1] + x[i3];
                s1 = y[i] - y[i2];
                y[i] = y[i] + y[i2];
                s2 = y[i1] - y[i3];
                y[i1] = y[i1] + y[i3];
                s3 = r1 - s2;
                r1 = r1 + s2;
                s2 = r2 - s1;
                r2 = r2 + s1;
                x[i2] = r1 * cc1 - s2 * ss1;
                y[i2] =-s2 * cc1 - r1 * ss1;
                x[i3] = s3 * cc3 + r2 * ss3;
                y[i3] = r2 * cc3 - s3 * ss3;
              }
            is = 2 * id - n2 + j;
            id = 4 * id;
          }
        while ( is < (n-1) );
      }
  }

is = 0;
id = 4;
do {
      for (i=is;i<n;i=i+id)
        { i1 = i + 1;
          r1 = x[i];
          r2 = y[i];
          x[i] = r1 + x[i1];
          y[i] = r2 + y[i1];
          x[i1] = r1 - x[i1];
          y[i1] = r2 - y[i1];
        }
      is = 2 * id - 2;
      id = 4 * id;
    } while ( is < (n-1) );
n1 = n-1;
```

```
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { r1 = x[j];
          s1 = y[j];
          x[j] = x[i];
          y[j] = y[i];
          x[i] = r1;
          y[i] = s1;
        }
      k = n/2;
      while ( k < (j+1) )
        { j = j-k;
          k = k/2;
        }
      j = j+k;
    }
}
```

五、例 题

设输入序列 $x(i)$ 为

$$x(i) = Q^i \quad , \quad i = 0,1,\cdots,n-1$$

其离散傅立叶变换为

$$X(k) = \frac{1 - Q^n}{1 - QW^k} \quad , \quad k = 0,1,\cdots,n-1$$

这里 $W = e^{-j\frac{2\pi}{n}}$，选取 $Q=0.9+j\,0.3$，$n=32$，用分裂基快速算法计算离散傅立叶变换 $X(k)$。

主函数程序(文件名:srfft.m):

```
#include "math.h"
#include "srfft.c"
main()
{ int i,j,n;
  double a1,a2,x[32],y[32];
  n = 32;
  a1 = 0.9;
  a2 = 0.3;
  x[0] = 1.0;
  y[0] = 0.0;
  for (i=1;i<n;i++)
```

```
  { x[i] = a1 * x[i-1] - a2 * y[i-1];
    y[i] = a2 * x[i-1] + a1 * y[i-1];
  }
srfft(x,y,n);
printf("\n DISCRETE FOURIER TRANSFORM\n");
for (i=0;i<n/2;i++)
  { for (j=0;j<2;j++)
      printf(" %10.7f + J %10.7f",x[2*i+j],y[2*i+j]);
    printf("\n");
  }
}
```

运行结果：

离散傅立叶变换 $X(k)(k=0,1,\cdots,31)$ 为

```
0.6939728  + J   3.4997157      2.7922679  + J   8.0504557
9.4029646  + J  -9.1350136      1.8664455  + J  -3.8338328
1.1318227  + J  -2.2341573      0.9047939  + J  -1.5346293
0.7995572  + J  -1.1396094      0.7396056  + J  -0.8823144
0.7008616  + J  -0.6985654      0.6735758  + J  -0.5584785
0.6531094  + J  -0.4462450      0.6369913  + J  -0.3526891
0.6237884  + J  -0.2720860      0.6126129  + J  -0.2006419
0.6028833  + J  -0.1357032      0.5942004  + J  -0.0753137
0.5862775  + J  -0.0179492      0.5788996  + J   0.0376517
0.5718985  + J   0.0926070      0.5651358  + J   0.1479833
0.5584921  + J   0.2048808      0.5518591  + J   0.2645222
0.5451336  + J   0.3283649      0.5382144  + J   0.3982571
0.5310015  + J   0.4766778      0.5234037  + J   0.5671323
0.5153625  + J   0.6748500      0.5069258  + J   0.8081015
0.4984670  + J   0.9809063      0.4913894  + J   1.2192074
0.4907322  + J   1.5770820      0.5173540  + J   2.1888329
```

§1.5　实序列快速傅立叶变换(一)

一、功　能

计算实序列的快速傅立叶变换

二、方法简介

实序列 $x(n)$ 的离散傅立叶变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},\quad k=0,1,\cdots,N-1$$

上式可用复序列 FFT 算法进行计算。但考虑到 $x(n)$是实数，为进一步提高计算效率，需要对按时间抽取的基 2 复序列 FFT 算法进行一定的修改。

如果序列 $x(n)$是实数，那么其傅立叶变换 $X(k)$一般是复数，但其实部是偶对称，虚部是奇对称，即 $X(k)$具有如下共轭对称性：$X(0)$和 $X(N/2)$都是实数，且有

$$X(k)=X^*(N-k),\quad 1\leqslant k\leqslant\frac{N}{2}-1$$

在计算离散傅立叶变换时，利用这种共轭对称性，我们就可以不必计算与存储 $X(k)$（$N/2+1\leqslant k\leqslant N-1$）以及 $X(0)$和 $X(N/2)$的虚部，而仅需计算 $X(0)$到 $X(N/2)$即可。此处我们选择的是计算 $X(0)$到 $X(N/4)$和 $X(N/2)$到 $X(3N/4)$，这样做可以恰好利用复序列 FFT 算法的前$(N/4)+1$ 个复数蝶形。这就是按时间抽取的基 2 实序列 FFT 算法，它比复序列 FFT 算法大约可减少一半的运算量和存储量。

三、使用说明

1. 子函数语句

void rfft(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果的前$\frac{n}{2}+1$ 个值，其存储顺序为[Re(0),Re(1),…,Re($\frac{n}{2}$),Im($\frac{n}{2}-1$),…,Im(1)]。其中 Re(0)=$X(0)$,Re($\frac{n}{2}$)= $X(\frac{n}{2})$。根据 $X(k)$的共轭对称性，很容易写出后半部分的值。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:srfft.c)

```
#include "math.h"
void rfft(x,n)
int n;
double x[];
{ int i,j,k,m,i1,i2,i3,i4,n1,n2,n4;
  double a,e,cc,ss,xt,t1,t2;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2 * j;
      if (j==n) break;
```

```
    }
  n1 = n-1;
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { xt = x[j];
          x[j] = x[i];
          x[i] = xt;
        }
      k = n/2;
      while ( k < (j+1) )
        { j = j - k;
          k = k/2;
        }
      j = j + k;
    }
  for (i=0;i<n;i+=2)
    { xt = x[i];
      x[i] = xt + x[i+1];
      x[i+1] = xt - x[i+1];
    }
  n2=1;
  for (k=2;k<=m;k++)
    { n4 = n2;
      n2 = 2*n4;
      n1 = 2*n2;
      e = 6.28318530718/n1;
      for (i=0;i<n;i+=n1)
        { xt = x[i];
          x[i] = xt + x[i+n2];
          x[i+n2] = xt - x[i+n2];
          x[i+n2+n4] = -x[i+n2+n4];
          a=e;
          for (j=1;j<=(n4-1);j++)
            { i1 = i + j;
              i2 = i - j + n2;
              i3 = i + j + n2;
              i4 = i - j + n1;
              cc = cos(a);
```

```
                ss = sin(a);
                a = a + e;
                t1 = cc * x[i3] + ss * x[i4];
                t2 = ss * x[i3] - cc * x[i4];
                x[i4] = x[i2] - t2;
                x[i3] = -x[i2] - t2;
                x[i2] = x[i1] - t1;
                x[i1] = x[i1] + t1;
              }
          }
      }
  }
```

五、例　题

设信号 $x(t)$ 为

$$x(t)=\begin{cases}0\,, & t<0.1\\ \exp\left[-\dfrac{t-0.1}{0.15}\right]\sin\left[\dfrac{2\pi(t-0.1)}{0.16}\right], & 0.1\leqslant t<0.64\end{cases}$$

取 $n=64$，采样间隔 $\Delta t=0.01$，则输入序列为

$$x(i)=\begin{cases}0\,, & i=0,1,\cdots,9\\ \exp\left[-\dfrac{i-10}{15}\right]\sin\left[\dfrac{2\pi(i-10)}{16}\right], & i=10,11,\cdots,63\end{cases}$$

先用实序列快速傅立叶变换算法计算出前 $\frac{n}{2}+1$ 点的 DFT，然后根据 $X(k)$ 的共轭对称性，得到 $X(k)$ 的后半部分的值。

主函数程序(文件名:rfft.m):

```
#include "math.h"
#include "rfft.c"
main()
{ int i,n;
  double x[64];
  n=64;
  for (i=0;i<10;i++)
    { x[i]=0.;}
  for (i=10;i<n;i++)
    { x[i]=exp(-(i-10)/15.0) * sin(6.2831853 * (i-10)/16.0);}
  rfft(x,n);
  printf("\n DISCRETE FOURIER TRANSFORM\n");
  printf(" %10.7f ",x[0]);
```

```
    printf(" %10.7f + J %10.7f\n",x[1],x[n-1]);
    for (i=2;i<n/2;i+=2)
      { printf(" %10.7f + J %10.7f",x[i],x[n-i]);
        printf(" %10.7f + J %10.7f",x[i+1],x[n-i-1]);
        printf("\n");
      }
    printf(" %10.7f ",x[n/2]);
    printf(" %10.7f + J %10.7f\n",x[n/2-1],-x[n/2+1]);
    for (i=2;i<n/2;i+=2)
      { printf(" %10.7f + J %10.7f",x[n/2-i],-x[n/2+i]);
        printf(" %10.7f + J %10.7f",x[n/2-i-1],-x[n/2+i+1]);
        printf("\n");
      }
  }
```

运行结果：

离散傅立叶变换 $X(k)(k=0,1,\cdots,63)$为

2.4716438			1.2740712	+ J	−2.2911342
−1.7773741	+ J	−2.6344206	−4.5176231	+ J	1.3403590
4.6976010	+ J	5.5565690	1.5895801	+ J	−3.3203257
−1.4488674	+ J	−1.3381975	−1.2146800	+ J	0.4119618
−0.2166592	+ J	0.8755663	0.4458803	+ J	0.4765451
0.4785094	+ J	−0.1037752	0.1151100	+ J	−0.3805316
−0.2368093	+ J	−0.2597506	−0.3160070	+ J	0.0427139
−0.1370433	+ J	0.2443001	0.0975637	+ J	0.2129842
0.1916384	+ J	0.0250160	0.0997018	+ J	−0.1391219
−0.0695481	+ J	−0.1529382	−0.1656937	+ J	−0.0313017
−0.1211163	+ J	0.1050731	0.0082553	+ J	0.1412694
0.1047725	+ J	0.0591762	0.0916960	+ J	−0.0600073
−0.0089390	+ J	−0.1129893	−0.1042640	+ J	−0.0613241
−0.1135680	+ J	0.0425398	−0.0345115	+ J	0.1059031
0.0601108	+ J	0.0761079	0.0878926	+ J	−0.0173288
0.0270939	+ J	−0.0914649	−0.0669673	+ J	−0.0825079
−0.1112558			−0.0669673	+ J	0.0825079
0.0270939	+ J	0.0914649	0.0878926	+ J	0.0173288
0.0601108	+ J	−0.0761079	−0.0345115	+ J	−0.1059031
−0.1135680	+ J	−0.0425398	−0.1042640	+ J	0.0613241
−0.0089390	+ J	0.1129893	0.0916960	+ J	0.0600073
0.1047725	+ J	−0.0591762	0.0082553	+ J	−0.1412694

−0.1211163	+ J	−0.1050731	−0.1656937	+ J	0.0313017
−0.0695481	+ J	0.1529382	0.0997018	+ J	0.1391219
0.1916384	+ J	−0.0250160	0.0975637	+ J	−0.2129842
−0.1370433	+ J	−0.2443001	−0.3160070	+ J	−0.0427139
−0.2368093	+ J	0.2597506	0.1151100	+ J	0.3805316
0.4785094	+ J	0.1037752	0.4458803	+ J	−0.4765451
−0.2166592	+ J	−0.8755663	−1.2146800	+ J	−0.4119618
−1.4488674	+ J	1.3381975	1.5895801	+ J	3.3203257
4.6976010	+ J	−5.5565690	−4.5176231	+ J	−1.3403590
−1.7773741	+ J	2.6344206	1.2740712	+ J	2.2911342

§1.6 实序列快速傅立叶变换(二)

一、功　能

用 N 点复序列快速傅立叶变换来计算 $2N$ 点实序列的离散傅立叶变换

二、方法简介

假设 $x(n)$是长度为 $2N$ 的实序列,其离散傅立叶变换为

$$X(k) = \sum_{n=0}^{2N-1} x(n) W_{2N}^{nk}, \quad k = 0,1,\cdots,2N-1$$

为有效地计算傅立叶变换 $X(k)$,我们将 $x(n)$分为偶数组和奇数组,形成两个新序列 $f(n)$和 $g(n)$,即

$$\begin{cases} f(n) = x(2n) \\ g(n) = x(2n+1), \quad n = 0,1,\cdots,N-1 \end{cases}$$

然后将 $f(n)$和 $g(n)$组成一个复序列 $h(n)$

$$h(n) = f(n) + jg(n), \quad n = 0,1,\cdots,N-1$$

用 FFT 计算 $h(n)$的 N 点傅立叶变换 $H(k)$,并且 $H(k)$可表示为

$$H(k) = F(k) + jG(k), \quad k = 0,1,\cdots,N-1$$

由上容易推出

$$\begin{cases} F(k) = \dfrac{1}{2}[H(k) + H^*(N-k)] \\ G(k) = -\dfrac{j}{2}[H(k) - H^*(N-k)] \end{cases}$$

求得 $F(k)$和 $G(k)$后,利用下面的蝶形运算计算 $x(n)$的离散傅立叶变换 $X(k)$

$$\begin{cases} X(k) = F(k) + W_{2N}^k G(k) \\ X(k+N) = F(k) - W_{2N}^k G(k), \quad k = 0,1,\cdots,N-1 \end{cases}$$

这种实序列 FFT 算法比相同长度的复序列 FFT 算法大约可减少一半的运算量。

三、使用说明

1. 子函数语句

```
void r1fft(x,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为n。开始时存放要变换的实数据，最后存放变换结果的前$\frac{n}{2}+1$个值，其存储顺序为［Re（0），Re（1），…，Re（$\frac{n}{2}$），Im（$\frac{n}{2}-1$），…，Im（1）］。其中Re（0）$=X$（0），Re（$\frac{n}{2}$）$=X$（$\frac{n}{2}$）。根据$X(k)$的共轭对称性，很容易写出后半部分的值。

n ——整型变量。数据长度，必须是2的整数次幂，即$n=2^m$。

四、子函数程序(文件名:r1fft.c)

```
#include "stdlib.h"
#include "math.h"
#include "fft.c"
void r1fft(x,n)
int n;
double x[];
{ int i,n1;
  double a,c,e,s,fr,fi,gr,gi,*f,*g;
  f = malloc(n/2*sizeof(double));
  g = malloc(n/2*sizeof(double));
  n1 = n/2;
  e = 3.141592653589793/n1;
  for (i=0;i<n1;i++)
    { f[i] = x[2*i];
      g[i] = x[2*i+1];
    }
  fft(f,g,n1,1);
  x[0] = f[0] + g[0];
  x[n1] = f[0] - g[0];
  for (i=1;i<n1;i++)
    { fr = (f[i]+f[n1-i])/2;
      fi = (g[i]-g[n1-i])/2;
      gr = (g[n1-i]+g[i])/2;
```

```
        gi = (f[n1--i]--f[i])/2;
        a = i*e;
        c = cos(a);
        s = sin(a);
        x[i] = fr + c*gr + s*gi;
        x[n-i] = fi + c*gi - s*gr;
      }
  free(f);
  free(g);
}
```

五、例　题

设信号 $x(t)$ 为

$$x(t)=\begin{cases}0, & t<0.1\\ \exp\left[-\dfrac{t-0.1}{0.15}\right]\sin\left[\dfrac{2\pi(t-0.1)}{0.16}\right], & 0.1\leqslant t<0.64\end{cases}$$

取 $n=64$，采样间隔 $\Delta t=0.01$，则输入序列为

$$x(i)=\begin{cases}0, & i=0,1,\cdots,9\\ \exp\left[-\dfrac{i-10}{15}\right]\sin\left[\dfrac{2\pi(i-10)}{16}\right], & i=10,11,\cdots,63\end{cases}$$

先用实序列快速傅立叶变换算法计算出前 $\dfrac{N}{2}+1$ 点的 DFT，然后画出其频谱图。

主函数程序（文件名：r1fft.m）：

```
#include "stdio.h"
#include "math.h"
#include "r1fft.c"
main()
{ int i,n; double x[64];
  FILE *fp;
  n=64;
  for (i=0;i<10;i++)
    { x[i]=0.;. }
  for (i=10;i<n;i++)
    { x[i]=exp(-(i-10)/15.0)*sin(6.2831853*(i-10)/16.0);. }
  r1fft(x,n);
  printf("\n DISCRETE FOURIER TRANSFORM\n");
  printf(" %10.7f ",x[0]);
  printf(" %10.7f + J %10.7f\n",x[1],x[n-1]);
  for (i=2;i<n/2;i+=2)
```

```
    { printf(" %10.7f + J %10.7f",x[i],x[n-i]);
      printf(" %10.7f + J %10.7f",x[i+1],x[n-i-1]);
      printf("\n");
    }
  printf(" %10.7f ",x[n/2]);
  for (i=1;i<n/2;i++)
    { x[i]=sqrt(x[i] * x[i]+x[n-i] * x[n-i]); }
  x[n/2]=fabs(x[n/2]);
  fp=fopen("r1fft.dat","w");
  for (i=0;i<=n/2;i++)
    { fprintf(fp,"%d %f\n",i,x[i]); }
  close(fp);
}
```

输入序列 $x(n)$ 的频谱如图 2-1-1 所示。

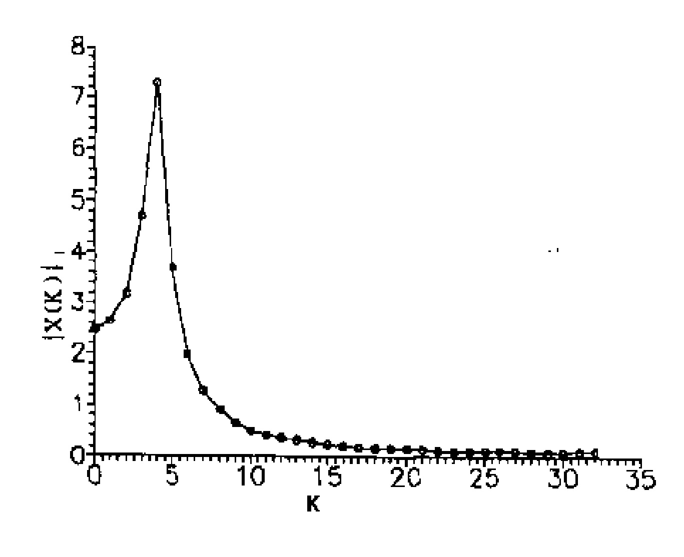


图 2—1—1　序列 $x(i)$ 的频谱

运行结果：

离散傅立叶变换 $X(k)(k=0,1,\cdots,32)$ 为

2.4716438			1.2740712	+J	−2.2911342
−1.7773741	+J	−2.6344206	−4.5176231	+J	1.3403590
4.6976010	+J	5.5565690	1.5895801	+J	−3.3203257
−1.4488674	+J	−1.3381975	−1.2146800	+J	0.4119618
−0.2166592	+J	0.8755663	0.4458803	+J	0.4765451
0.4785094	+J	−0.1037752	0.1151100	+J	−0.3805316
−0.2368093	+J	−0.2597506	−0.3160070	+J	0.0427139
−0.1370433	+J	0.2443001	0.0975637	+J	0.2129842
0.1916384	+J	0.0250160	0.0997018	+J	−0.1391219

−0.0695481	+ J	−0.1529382	−0.1656937	+ J	−0.0313017
−0.1211163	+ J	0.1050731	0.0082553	+ J	0.1412694
0.1047725	+ J	0.0591762	0.0916960	+ J	−0.0600073
−0.0089390	+ J	−0.1129893	−0.1042640	+ J	−0.0613241
−0.1135680	+ J	0.0425398	−0.0345115	+ J	0.1059031
0.0601108	+ J	0.0761079	0.0878926	+ J	−0.0173288
0.0270939	+ J	−0.0914649	−0.0669673	+ J	−0.0825079
−0.1112558					

§1.7 用一个 N 点复序列的 FFT 同时计算两个 N 点实序列离散傅立叶变换

一、功　能

用一个 N 点复序列快速傅立叶变换算法来同时计算两个 N 点实序列的离散傅立叶变换

二、方法简介

假设 $x(n)$ 与 $y(n)$ 都是长度为 N 的实序列，为计算其离散傅立叶变换 $X(k)$ 与 $Y(k)$，我们将 $x(n)$ 与 $y(n)$ 组合成一个复数序列 $h(n)$

$$h(n) = x(n) + j\,y(n)$$

通过 FFT 运算可以获得 $h(n)$ 的离散傅立叶变换 $H(k)$，$H(k)$ 可表示为

$$H(k) = X(k) + j\,Y(k)$$

根据求得的 $H(k)$，并利用 DFT 的奇偶共轭性，我们得到 $X(k)$ 和 $Y(k)$ 为

$$\begin{cases} X(k) = \dfrac{1}{2}[H(k) + H^*(N-k)] \\ Y(k) = -\dfrac{j}{2}[H(k) - H^*(N-k)] \end{cases}$$

三、使用说明

1. 子函数语句

```
void r2fft(x,y,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放第一个实序列，最后存放变换结果的前 $\frac{n}{2}+1$ 个值，其存储顺序为 [Re(0), Re(1), …, Re($\frac{n}{2}$), Im($\frac{n}{2}-1$), …,

Im(1)]。其中 Re(0)=$X(0)$,Re($\frac{n}{2}$)=$X(\frac{n}{2})$。

y —— 双精度实型一维数组,长度为 n。开始时存放第二个实序列,最后存放变换结果的前$\frac{n}{2}+1$个值,其存储顺序为[Re(0),Re(1),…,Re($\frac{n}{2}$),Im($\frac{n}{2}-1$),…,Im(1)]。其中 Re(0)=$Y(0)$,Re($\frac{n}{2}$)= $Y(\frac{n}{2})$。

根据 $X(k)$和 $Y(k)$的共轭对称性,很容易给出它们后半部分的值。

n —— 整型变量。数据长度,必须是 2 的整数次幂,即 $n=2^m$。

四、子函数程序(文件名:r2fft.c)

```
#include "fft.c"
void r2fft(x,y,n)
int n;
double x[],y[];
{ int i,n1;
  double tr,ti;
  n1 = n/2;
  fft(x,y,n,1);
  for (i=1;i<n1;i++)
    { tr = (x[i] + x[n-i])/2;
      ti = (y[i] - y[n-i])/2;
      y[i] = (y[n-i] + y[i])/2;
      y[n-i] = (x[n-i] - x[i])/2;
      x[i] = tr;
      x[n-i] = ti;
    }
}
```

五、例 题

取 $n=64$,第一个实序列 $x(i)$为

$$x(i) = e^{-0.1i} \quad, i = 0,1,\cdots,63$$

第二个实序列 $y(i)$为

$$y(i) = \begin{cases} 0\,, & i = 0,1,\cdots,9 \\ \exp\left[-\frac{i-10}{15}\right]\sin\left[\frac{2\pi(i-10)}{16}\right], & i = 10,11,\cdots,63 \end{cases}$$

用复序列快速傅立叶变换算法,同时计算出两个实序列 $x(i)$和 $y(i)$的前$\frac{n}{2}+1$点的 DFT。

主函数程序(文件名:r2fft.m):

```
#include "math.h"
#include "r2fft.c"
main()
{ int i,n;
  double x[64],y[64];
  n=64;
  for (i=0;i<n;i++)
    { x[i]=exp(-0.1*i); }
  for (i=0;i<10;i++)
    { y[i]=0.; }
  for (i=10;i<n;i++)
    { y[i]=exp(-(i-10)/15.0)*sin(6.2831853*(i-10)/16.0); }
  r2fft(x,y,n);
  printf("\n DISCRETE FOURIER TRANSFORM X(k):\n");
  printf(" %10.7f ",x[0]);
  printf(" %10.7f + J %10.7f\n",x[1],x[n-1]);
  for (i=2;i<n/2;i+=2)
    { printf(" %10.7f + J %10.7f",x[i],x[n-i]);
      printf(" %10.7f + J %10.7f",x[i+1],x[n-i-1]);
      printf("\n");
    }
  printf(" %10.7f\n",x[n/2]);
  printf("\n DISCRETE FOURIER TRANSFORM Y(k):\n");
  printf(" %10.7f ",y[0]);
  printf(" %10.7f + J %10.7f\n",y[1],y[n-1]);
  for (i=2;i<n/2;i+=2)
    { printf(" %10.7f + J %10.7f",y[i],y[n-i]);
      printf(" %10.7f + J %10.7f",y[i+1],y[n-i-1]);
      printf("\n");
    }
  printf(" %10.7f\n",y[n/2]);
}
```

运行结果：

第一个序列的离散傅立叶变换 $X(k)(k=0,1,\cdots,33)$为

```
10.4908717                          5.5911247   + J   -4.9826806
 2.5636802   + J   -4.0209561       1.5394558   + J   -3.0147654
 1.1155068   + J   -2.3546952       0.9054008   - J   -1.9117744
 0.7872972   + J   -1.5980884       0.7146930   + J   -1.3649922
```

0.6670128 + J −1.1848575 　 0.6340820 + J −1.0411541

0.6104206 + J −0.9234818 　 0.5928714 + J −0.8250034

0.5795139 + J −0.7410523 　 0.5691270 + J −0.6683394

0.5609044 + J −0.6044814 　 0.5542970 + J −0.5477097

0.5489204 + J −0.4966837 　 0.5444994 + J −0.4503681

0.5408327 + J −0.4079494 　 0.5377708 + J −0.3687789

0.5352005 + J −0.3323314 　 0.5330355 + J −0.2981764

0.5312088 + J −0.2659559 　 0.5296683 + J −0.2353688

0.5283730 + J −0.2061583 　 0.5272906 + J −0.1781029

0.5263954 + J −0.1510087 　 0.5256674 + J −0.1247038

0.5250910 + J −0.0990333 　 0.5246542 + J −0.0738559

0.5243482 + J −0.0490400 　 0.5241669 + J −0.0244613

0.5241069

第二个序列的离散傅立叶变换 $Y(k)(k=0,1,\cdots,33)$ 为

2.4716438 　 1.2740712 + J −2.2911342

−1.7773741 + J −2.6344206 　 −4.5176231 + J 1.3403590

4.6976010 + J 5.5565690 　 1.5895801 + J −3.3203257

−1.4488674 + J −1.3381975 　 −1.2146800 + J 0.4119618

−0.2166592 + J 0.8755663 　 0.4458803 + J 0.4765451

0.4785094 + J −0.1037752 　 0.1151100 + J −0.3805316

−0.2368093 + J −0.2597506 　 −0.3160070 + J 0.0427139

−0.1370433 + J 0.2443001 　 0.0975637 + J 0.2129842

0.1916384 + J 0.0250160 　 0.0997018 + J −0.1391219

−0.0695481 + J −0.1529382 　 −0.1656937 + J −0.0313017

−0.1211163 + J 0.1050731 　 0.0082553 + J 0.1412694

0.1047725 + J 0.0591762 　 0.0916960 + J −0.0600073

−0.0089390 + J −0.1129893 　 −0.1042640 + J −0.0613241

−0.1135680 + J 0.0425398 　 −0.0345115 + J 0.1059031

0.0601108 + J 0.0761079 　 0.0878926 + J −0.0173288

0.0270939 + J −0.0914649 　 −0.0669673 + J −0.0825079

−0.1112558

§1.8 共轭对称序列的快速傅立叶反变换

一、功　能

计算共轭对称复序列的快速傅立叶反变换，其变换结果是实数。

二、方法简介

序列 $x(n)$ 的离散傅立叶变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},\quad k=0,1,\cdots,N-1$$

序列 $X(k)$ 的离散傅立叶反变换为

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)W_N^{-nk},\quad n=0,1,\cdots,N-1$$

共轭对称复序列的傅立叶反变换，可用复序列快速傅立叶反变换算法进行计算。但考虑到 $X(k)$ 是共轭对称的，其傅立叶反变换 $x(n)$ 是实数，因此，为进一步提高计算效率，需要对一般的复序列 IFFT 算法进行一定的修改。

共轭对称序列 $X(k)$ 具有如下性质：$X(0)$ 和 $X(\frac{N}{2})$ 都是实数，且有

$$X(k)=X^*(N-k),\quad 1\leqslant k\leqslant\frac{N}{2}-1$$

即 $X(k)$ 的实部是偶对称，虚部是奇对称。在计算傅立叶反变换时，利用这种共轭对称性，我们就可以不必计算和存储 $X(k)(\frac{N}{2}+1\leqslant k\leqslant N-1)$ 以及 $X(0)$ 和 $X(\frac{N}{2})$ 的虚部，这比一般形式的快速傅立叶反变换算法大约可减少一半的运算量和存储量。具体计算时采用的是分裂基算法

三、使用说明

1. 子函数语句

void irfft(x,n)

2. 形参说明

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

x ——双精度实型一维数组，长度为 n。开始时存放具有共轭对称性的复序列 $X(k)$ 的前 $\frac{n}{2}+1$ 个值，其存储顺序为 $[Re(0),Re(1),\cdots,Re(\frac{n}{2}),Im(\frac{n}{2}-1),\cdots,Im(1)]$，其中 $Re(0)=X(0)$，$Re(\frac{n}{2})=X(\frac{n}{2})$。最后存放变换结果 $x(i)$ $(i=0,1,\cdots,n-1)$，这里 $x(i)$ 是实数。

四、子函数程序(文件名：irfft.c)

```
#include "math.h"
void irfft(x,n)
int n;
double x[];
{ int i,j,k,m,i1,i2,i3,i4,i5,i6,i7,i8,n2,n4,n8,id,is;
```

```
double a,e,a3,t1,t2,t3,t4,t5,cc1,cc3,ss1,ss3;
for (j=1,i=1;i<16;i++)
  { m = i;
    j = 2*j;
    if (j==n) break;
  }
n2 = 2*n;
for (k=1;k<m;k++)
  { is = 0;
    id = n2;
    n2 = n2/2;
    n4 = n2/4;
    n8 = n4/2;
    e = 6.28318530718/n2;
    do
      { for (i=is;i<n;i+=id)
          { i1 = i;
            i2 = i1 + n4;
            i3 = i2 + n4;
            i4 = i3 + n4;
            t1 = x[i1] - x[i3];
            x[i1] = x[i1] + x[i3];
            x[i2] = 2*x[i2];
            x[i3] = t1 - 2*x[i4];
            x[i4] = t1 + 2*x[i4];
            if ( n4 == 1 ) continue;
            i1 += n8;
            i2 += n8;
            i3 += n8;
            i4 += n8;
            t1 = (x[i2] - x[i1])/sqrt(2.0);
            t2 = (x[i4] + x[i3])/sqrt(2.0);
            x[i1] = x[i1] + x[i2];
            x[i2] = x[i4] - x[i3];
            x[i3] = 2*( -t2 - t1 );
            x[i4] = 2*( -t2 + t1 );
          }
        is = 2*id-n2;
```

```
      id = 4 * id;
    } while ( is < (n-1) );
  a = e;
  for (j=1;j<n8;j++)
    { a3 = 3 * a;
      cc1 = cos(a);
      ss1 = sin(a);
      cc3 = cos(a3);
      ss3 = sin(a3);
      a = (j + 1) * e;
      is = 0;
      id = 2 * n2;
      do
        { for (i=is;i<=(n-1);i=i+id)
            { i1 = i - j;
              i2 = i1 + n4;
              i3 = i2 + n4;
              i4 = i3 + n4;
              i5 = i + n4 - j;
              i6 = i5 + n4;
              i7 = i6 + n4;
              i8 = i7 + n4;
              t1 = x[i1] - x[i6];
              x[i1] = x[i1] + x[i6];
              t2 = x[i5] - x[i2];
              x[i5] = x[i2] + x[i5];
              t3 = x[i8] + x[i3];
              x[i6] = x[i8] - x[i3];
              t4 = x[i4] + x[i7];
              x[i2] = x[i4] - x[i7];
              t5 = t1 - t4;
              t1 = t1 + t4;
              t4 = t2 - t3;
              t2 = t2 + t3;
              x[i3] = t5 * cc1 + t4 * ss1;
              x[i7] = -t4 * cc1 + t5 * ss1;
              x[i4] = t1 * cc3 - t2 * ss3;
              x[i8] = t2 * cc3 + t1 * ss3;
```

```
                }
            is = 2 * id - n2;
            id = 4 * id;
          }
        while ( is < (n -1) );
      }
  }
is = 0;
id = 4;
do
  { for (i=is;i<n;i=i+id)
      { i1 = i + 1;
        t1 = x[i];
        x[i] = t1 + x[i1];
        x[i1] = t1 - x[i1];
      }
    is = 2 * id - 2;
    id = 4 * id;
  } while ( is < (n-1) );
for (j=0,i=0;i<(n-1);i++)
  { if ( i < j )
      { t1 = x[j];
        x[j] = x[i];
        x[i] = t1;
      }
    k = n/2;
    while ( k < ( j + 1 ) )
      { j = j - k;
        k = k/2;
      }
    j = j + k;
  }
for (i=0;i<n;i++)
  x[i] = x[i]/n;
}
```

五、例　题

设 $n=32$，输入序列为

$$x(i) = e^{-i/15} \sin \frac{2\pi i}{16} \quad , \quad i = 0,1,\cdots,n-1$$

先用实序列快速傅立叶变换算法计算其 DFT，然后再利用 $X(k)$的共轭对称性计算快速傅立叶反变换。

主函数程序(文件名:irfft.m)：

```
#include "math.h"
#include "rfft.c"
#include "irfft.c"
main()
{ int i,n;
  double x[32];
  n=32;
  for (i=0;i<n;i++)
    { x[i]=exp(-i/15.0) * sin(6.2831853 * i/16.0); }
  printf("\n Original Sequence\n");
  for (i=0;i<n;i+=4)
    { printf(" %10.7f    %10.7f",x[i],x[i+1]);
      printf(" %10.7f    %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
  rfft(x,n);
  printf("\n Discrete Fourier Transform\n");
  printf(" %10.7f ",x[0]);
  printf(" %10.7f + J %10.7f\n",x[1],x[n-1]);
  for (i=2; i<(n/2); i+=2)
    { printf(" %10.7f + J %10.7f",x[i],x[n-i]);
      printf(" %10.7f + J %10.7f",x[i+1],x[n-i-1]);
      printf("\n");
    }
  printf(" %10.7f ",x[n/2]);
  printf(" %10.7f + J %10.7f\n",x[n/2-1],-x[n/2+1]);
  for (i=2;i<n/2;i+=2)
    { printf(" %10.7f + J %10.7f",x[n/2-i],-x[n/2+i]);
      printf(" %10.7f + J %10.7f",x[n/2-i-1],-x[n/2+i+1]);
      printf("\n");
    }
  irfft(x,n);
  printf("\n Inverse Discrete Fourier Transform\n");
```

```
    for (i=0;i<n;i+=4)
      { printf(" %10.7f   %10.7f",x[i],x[i+1]);
        printf(" %10.7f   %10.7f",x[i+2],x[i+3]);
        printf("\n");
      }
  }
```

运行结果：

原始输入序列 $x(i)$

0.0000000	0.3580030	0.6188410	0.7564086
0.7659283	0.6619886	0.4739879	0.2399766
0.0000000	−0.2100211	−0.3630407	−0.4437442
−0.4493290	−0.3883531	−0.2780632	−0.1407814
−0.0000000	0.1232081	0.2129765	0.2603209
0.2635971	0.2278259	0.1631247	0.0825889
0.0000000	−0.0722796	−0.1249418	−0.1527163
−0.1546383	−0.1336532	−0.0956965	−0.0484504

序列 $x(i)$的离散傅立叶变换 $X(k)$

2.1530673	+ J		2.7226816	+ J	−0.5997604
0.5280594	+ J	−6.5643121	−1.5953456	+ J	−0.6529507
−0.7478809	+ J	−0.1639472	−0.4492625	+ J	−0.0678927
−0.3081588	+ J	−0.0351519	−0.2297345	+ J	−0.0206389
−0.1816295	+ J	−0.0131160	−0.1501732	+ J	−0.0087783
−0.1287311	+ J	−0.0060690	−0.1137598	+ J	−0.0042619
−0.1032356	+ J	−0.0029832	−0.0959504	+ J	−0.0020239
−0.0911696	+ J	−0.0012583	−0.0884555	+J	−0.0006038
−0.0875752			−0.0884555	+ J	0.0006038
−0.0911696	+ J	0.0012583	−0.0959504	+ J	0.0020239
−0.1032356	+ J	0.0029832	−0.1137598	+ J	0.0042619
−0.1287311	+ J	0.0060690	−0.1501732	+ J	0.0087783
−0.1816295	+ J	0.0131160	−0.2297345	+ J	0.0206389
−0.3081588	+ J	0.0351519	−0.4492625	+ J	0.0678927
−0.7478809	+ J	0.1639472	−1.5953456	+ J	0.6529507
0.5280594	+ J	6.5643121	2.7226816	+ J	0.5997604

序列 $X(k)$ 的离散傅立叶反变换

0.0000000	0.3580030	0.6188410	0.7564086
0.7659283	0.6619886	0.4739879	0.2399766
0.0000000	−0.2100211	−0.3630407	−0.4437442
−0.4493290	−0.3883531	−0.2780632	−0.1407814

-0.0000000	0.1232081	0.2129765	0.2603209
0.2635971	0.2278259	0.1631247	0.0825889
0.0000000	-0.0722796	0.1249418	-0.1527163
-0.1546383	-0.1336532	-0.0956965	-0.0484504

§1.9 素因子快速傅立叶变换

一、功　能

用素因子分解算法计算复序列的离散傅立叶变换。序列的长度是数集{2,3,4,5,7,8,9,16}中的一个或几个互素因子的乘积。

二、方法简介

序列 $x(n)$ 的离散傅立叶变换为

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, \quad k = 0,1,\cdots,N-1$$

若 N 可以分解为两两互素因子的乘积，即

$$N = N_1 \cdot N_2 \cdot \cdots \cdot N_M, \quad (N_i, N_j) = 1 \quad (i \neq j)$$

那么素因子快速傅立叶变换算法由以下步骤组成：

1. 一维 DFT 映射为多维 DFT

对输入下标 n 作多因子简单映射，输出下标 k 作多因子孙子定理映射

$$\begin{cases} n \equiv \sum_{i=1}^{M} \frac{N}{N_i} n_i \bmod N \\ k \equiv \sum_{i=1}^{M} \frac{N}{N_i} t_i k_i \bmod N \end{cases}, \quad (n_i, k_i = 0,1,\cdots,N_i - 1)$$

其中 t_i 满足

$$\sum_{i=1}^{M} \frac{N}{N_i} t_i \equiv 1 \bmod N$$

并记

$$x(n_1,n_2,\cdots,n_M) = x(\frac{N}{N_1}n_1 + \cdots + \frac{N}{N_M}n_M)$$

$$X(k_1,k_2,\cdots,k_M) = X(\frac{N}{N_1}t_1k_1 + \cdots + \frac{N}{N_M}t_Mk_M)$$

于是得到 M 维 DFT，即

$$X(k_1,k_2,\cdots,k_M) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \cdots \sum_{n_M=0}^{N_M-1} x(n_1,n_2,\cdots,n_M) W_{N_1}^{n_1k_1} W_{N_2}^{n_2k_2} \cdots W_{N_M}^{n_Mk_M}$$

$$(k_i = 0,1,\cdots,N_i - 1;\ i = 1,2,\cdots,M)$$

这样就将原来长度为 N 的一维 DFT 映射为长度分别为 N_1，N_2，…，N_M 的 M 维 DFT，

从而一维长 DFT 可用多维短 DFT 实现。下面将说明，这些短 DFT 可用短循环卷积来计算。

2. 用 Rader 算法将 DFT 转换为循环卷积

设 p 为素数，p 点 DFT 为

$$X(k)=\sum_{n=0}^{p-1}x(n)W_N^{nk},\quad k=0,1,\cdots,p-1$$

将 $n=0$ 和 $k=0$ 的点单独列出，则有

$$X(0)=\sum_{n=0}^{p-1}x(n),$$

$$X(k)=x(0)+\overline{X}(k),\quad k=1,2,\cdots,p-1$$

其中 $$\overline{X}(k)=\sum_{n=1}^{p-1}x(n)W_N^{nk},\quad k=1,2,\cdots,p-1$$

设

$$\begin{cases}n\equiv g^{-u}\bmod p\\ k\equiv g^{v}\bmod p\end{cases}$$

则上面的 DFT $\overline{X}(k)$就成为循环卷积

$$\overline{X}(g^v)=\sum_{u=0}^{p-2}x(g^{-u})W_N^{g^{v-u}},\quad v=0,1,\cdots,p-2$$

3. 用 Winograd 算法计算短卷积

三、使用说明

1. 子函数语句

```
void pft(x,y,n,m,ni)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换数据的实部，最后存放变换结果的实部。

y ——双精度实型一维数组，长度为 n。开始时存放要变换数据的虚部，最后存放变换结果的虚部。

n ——整型变量。数据长度。n 必须能分解为 m 个两两互素整数的乘积，即n=ni(0) * ni(1) * … * ni(m−1)。

m ——整型变量。素因子的个数。

ni ——整型一维数组，长度为 m。用于存放 m 个两两互素的素因子。

本程序提供可选择的素因子为 2,3,4,5,7,8,9,16。因此，可选择的最大 m=4，DFT 的最大长度 n=5040。

四、子函数程序(文件名:pft.c)

```
#include "math.h"
#include "stdlib.h"
void pft(x,y,n,m,ni)
int m,n,ni[];
double x[],y[];
{ int j,k,l,it,n1,n2,unsc,i[17];
  double c31,c32,c51,c52,c53,c54,c55;
  double c71,c72,c73,c74,c75,c76,c77,c78,c81;
  double c92,c93,c94,c95,c96,c97,c98;
  double c162,c163,c164,c165;
  double r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14,r15,r16;
  double s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16;
  double t,t0,t1,t2,t3,t4,t5,t6,t7,t8,t9;
  double *a,*b;
  c31=-0.86602540; c32=-1.5;
  c51= 0.95105652; c52=-1.53884180;
  c53=-0.36327126; c54= 0.55901699;
  c55=-1.25;
  c71=-1.16666667; c72=-0.79015647;
  c73= 0.055854267; c74= 0.7343022 ;
  c75= 0.44095855; c76=-0.34087293;
  c77= 0.53396936; c78= 0.87484229;
  c81= 0.70710678;
  c92= 0.93969262; c93=-0.17364818;
  c94= 0.76604444; c95=-0.5;
  c96=-0.34202014; c97=-0.98480775;
  c98=-0.64278761;
  c162= 0.38268343; c163= 1.30656297;
  c164= 0.54119610; c165= 0.92387953;

  for (k=0;k<m;k++)
    { n1=ni[k];
      n2=n/n1;
      for (j=0;j<n;j+=n1)
        { it=j;
          i[1]=j;
```

```
for (l=2;l<=n1;l++)
  { it=it+n2;
    if (it>(n-1)) it=it-n;
    i[l]=it;
  }

switch ( n1 )
  { case 2:
      { r1=x[i[1]];
        x[i[1]]=r1+x[i[2]];
        x[i[2]]=r1-x[i[2]];
        r1=y[i[1]];
        y[i[1]]=r1+y[i[2]];
        y[i[2]]=r1-y[i[2]];
        break;
      }
    case 3:
      { r1=x[i[2]]+x[i[3]];
        r2=(x[i[2]]-x[i[3]]) * c31;
        x[i[1]]=x[i[1]]+r1;
        r1 =x[i[1]]+r1 * c32;
        s1=y[i[2]]+y[i[3]];
        s2=(y[i[2]]-y[i[3]]) * c31;
        y[i[1]]=y[i[1]]+s1;
        s1 =y[i[1]]+s1 * c32;
        x[i[2]]=r1-s2;
        x[i[3]]=r1+s2;
        y[i[2]]=s1+r2;
        y[i[3]]=s1-r2;
        break;
      }
    case 4:
      { r1=x[i[1]]+x[i[3]];
        t1=x[i[1]]-x[i[3]];
        r2=x[i[2]]+x[i[4]];
        x[i[1]]=r1+r2;
        x[i[3]]=r1-r2;
        r1=y[i[1]]+y[i[3]];
```

```
       t2=y[i[1]]-y[i[3]];
       r2=y[i[2]]+y[i[4]];
       y[i[1]]=r1+r2;
       y[i[3]]=r1-r2;
       r1=x[i[2]]-x[i[4]];
       r2=y[i[2]]-y[i[4]];
       x[i[2]]=t1+r2;
       x[i[4]]=t1-r2;
       y[i[2]]=t2-r1;
       y[i[4]]=t2+r1;
       break;
     }
   case 5:
     { r1=x[i[2]]+x[i[5]];
       r4=x[i[2]]-x[i[5]];
       r3=x[i[3]]+x[i[4]];
       r2=x[i[3]]-x[i[4]];
       t = (r1-r3) * c54;
       r1= r1-r3;
       x[i[1]]=x[i[1]]+r1;
       r1 =x[i[1]]+r1 * c55;
       r3=r1-t;
       r1=r1+t;
       t =(r4+r2) * c51;
       r4=t+r4 * c52;
       r2=t+r2 * c53;

       s1=y[i[2]]+y[i[5]];
       s4=y[i[2]]-y[i[5]];
       s3=y[i[3]]+y[i[4]];
       s2=y[i[3]]-y[i[4]];
       t = (s1-s3) * c54;
       s1= s1+s3;
       y[i[1]]=y[i[1]]+s1;
       s1 =y[i[1]]+s1 * c55;
       s3=s1-t;
       s1=s1+t;
       t =(s4+s2) * c51;
```

```
      s4=t+s4*c52;
      s2=t+s2*c53;

      x[i[2]]=r1+s2;
      x[i[5]]=r1-s2;
      x[i[3]]=r3-s4;
      x[i[4]]=r3+s4;
      y[i[2]]=s1-r2;
      y[i[5]]=s1+r2;
      y[i[3]]=s3+r4;
      y[i[4]]=s3-r4;
      break;
    }
case 7:
    { r1=x[i[2]]+x[i[7]];
      r6=x[i[2]]-x[i[7]];
      s1=y[i[2]]+y[i[7]];
      s6=y[i[2]]-y[i[7]];
      r2=x[i[3]]+x[i[6]];
      r5=x[i[3]]-x[i[6]];
      s2=y[i[3]]+y[i[6]];
      s5=y[i[3]]-y[i[6]];
      r3=x[i[4]]+x[i[5]];
      r4=x[i[4]]-x[i[5]];
      s3=y[i[4]]+y[i[5]];
      s4=y[i[4]]-y[i[5]];

      t3= (r1-r2)*c74;
      t = (r1-r3)*c72;
      r1= r1+r2+r3;
      x[i[1]]=x[i[1]]+r1;
      r1 =x[i[1]]+r1*c71;
      r2=(r3-r2)*c73;
      r3=r1-t +r2;
      r2=r1-r2-t3;
      r1=r1+t+t3;
      t =(r6-r5)*c78;
      t3=(r6+r4)*c76;
```

```
        r6=(r6+r5-r4)*c75;
        r5=(r5+r4)*c77;
        r4=r6-t3+r5;
        r5=r6-r5-t;
        r6=r6+t3+t;

        t3= (s1-s2)*c74;
        t = (s1-s3)*c72;
        s1= s1+s2+s3;
        y[i[1]]=y[i[1]]+s1;
        s1 =y[i[1]]+s1*c71;
        s2=(s3-s2)*c73;
        s3-s1-t +s2;
        s2=s1-s2-t3;
        s1=s1+t+t3;
        t =(s6-s5)*c78;
        t3=(s6+s4)*c76;
        s6=(s6+s5-s4)*c75;
        s5=(s5+s4)*c77;
        s4=s6-t3+s5;
        s5=s6-s5-t;
        s6=s6+t3+t;

        x[i[2]]=r3+s4;
        x[i[7]]=r3-s4;
        x[i[3]]=r1+s6;
        x[i[6]]=r1-s6;
        x[i[4]]=r2-s5;
        x[i[5]]=r2+s5;
        y[i[2]]=s3-r4;
        y[i[7]]=s3+r4;
        y[i[3]]=s1-r6;
        y[i[6]]=s1+r6;
        y[i[4]]=s2+r5;
        y[i[5]]=s2-r5;
        break;
      }
    case 8:
```

```
{ r1=x[i[1]]+x[i[5]];
  r2=x[i[1]]-x[i[5]];
  r3=x[i[2]]+x[i[8]];
  r4=x[i[2]]-x[i[8]];
  r5=x[i[3]]+x[i[7]];
  r6=x[i[3]]-x[i[7]];
  r7=x[i[4]]+x[i[6]];
  r8=x[i[4]]-x[i[6]];
  t1 = r1+r5;
  t2 = r1-r5;
  t3 = r3+r7;
  r3 =(r3-r7) * c81;
  x[i[1]] = t1+t3;
  x[i[5]] = t1-t3;
  t1 = r2+r3;
  t3 = r2-r3;
  s1 = r4-r8;
  r4 =(r4+r8) * c81;
  s2 = r4+r6;
  s3 = r4-r6;

  r1=y[i[1]]+y[i[5]];
  r2=y[i[1]]-y[i[5]];
  r3=y[i[2]]+y[i[8]];
  r4=y[i[2]]-y[i[8]];
  r5=y[i[3]]+y[i[7]];
  r6=y[i[3]]-y[i[7]];
  r7=y[i[4]]+y[i[6]];
  r8=y[i[4]]-y[i[6]];
  t4 = r1+r5;
  r1 = r1-r5;
  r5 = r3+r7;
  r3 =(r3-r7) * c81;
  y[i[1]] = t4+r5;
  y[i[5]] = t4-r5;
  r5 = r2+r3;
  r2 = r2-r3;
  r3 = r4-r8;
```

```
        r4 =(r4+r8) * c81;
        r7 = r4-r6;
        r4 = r4-r6;

        x[i[2]]=t1+r7;
        x[i[8]]=t1 -r7;
        x[i[3]]=t2+r3;
        x[i[7]]=t2-r3;
        x[i[4]]=t3+r4;
        x[i[6]]=t3-r4;
        y[i[2]]=r5-s2;
        y[i[8]]=r5+s2;
        y[i[3]]=r1 -s1;
        y[i[7]]=r1+s1;
        y[i[4]]=r2-s3;
        y[i[6]]=r2+s3;
        break;
      }
    case 9:
      { r1= x[i[2]]+x[i[9]];
        r2= x[i[2]]-x[i[9]];
        r3= x[i[3]]+x[i[8]];
        r4= x[i[3]]-x[i[8]];
        r5= x[i[4]]+x[i[7]];
        t8=(x[i[4]]-x[i[7]]) * c31;
        r7= x[i[5]]+x[i[6]];
        r8= x[i[5]]-x[i[6]];
        t0=x[i[1]]+r5;
        t7=x[i[1]]+r5 * c95;
        r5= r1+r3+r7;
        x[i[1]]=t0+r5;
        t5= t0+r5 * c95;
        t3=(r3-r7) * c92;
        r7=(r1-r7) * c93;
        r3=(r1-r3) * c94;
        t1=t7+t3+r3;
        t3=t7-t3-r7;
        t7=t7+r7-r3;
```

```
t6=(r2-r4+r8)*c31;
t4=(r4+r8)*c96;
r8=(r2-r8)*c97;
r2=(r2+r4)*c98;
t2=t8+t4+r2;
t4=t8-t4-r8;
t8=t8+r8-r2;

r1= y[i[2]]+y[i[9]];
r2= y[i[2]]-y[i[9]];
r3= y[i[3]]+y[i[8]];
r4= y[i[3]]-y[i[8]];
r5= y[i[4]]+y[i[7]];
r6=(y[i[4]]-y[i[7]])*c31;
r7= y[i[5]]+y[i[6]];
r8= y[i[5]]-y[i[6]];
t0= y[i[1]]+r5;
t9= y[i[1]]+r5*c95;
r5= r1+r3+r7;
y[i[1]]=t0+r5;
r5= t0+r5*c95;
t0=(r3-r7)*c92;
r7=(r1-r7)*c93;
r3=(r1-r3)*c94;
r1=t9+t0+r3;
t0=t9-t0-r7;
r7=t9+r7-r3;
r9=(r2-r4+r8)*c31;
r3=(r4+r8)*c96;
r8=(r2-r8)*c97;
r4=(r2+r4)*c98;
r2=r6+r3+r4;
r3=r6-r8-r3;
r8=r6+r8-r4;

x[i[2]]=t1-r2;
x[i[9]]=t1+r2;
x[i[3]]=t3+r3;
```

```
        x[i[8]]=t3-r3;
        x[i[4]]=t5-r9;
        x[i[7]]=t5+r9;
        x[i[5]]=t7-r8;
        x[i[6]]=t7+r8;
        y[i[2]]=r1+t2;
        y[i[9]]=r1-t2;
        y[i[3]]=t0-t4;
        y[i[8]]=t0+t4;
        y[i[4]]=r5+t6;
        y[i[7]]=r5-t6;
        y[i[5]]=r7+t8;
        y[i[6]]=r7-t8;
        break;
      }
    case 16:
      { r1 =x[i[1]]+x[i[9]];
        r2 =x[i[1]]-x[i[9]];
        r3 =x[i[2]]+x[i[10]];
        r4 =x[i[2]]-x[i[10]];
        r5 =x[i[3]]+x[i[11]];
        r6 =x[i[3]]-x[i[11]];
        r7 =x[i[4]]+x[i[12]];
        r8 =x[i[4]]-x[i[12]];
        r9 =x[i[5]]+x[i[13]];
        r10=x[i[5]]-x[i[13]];
        r11=x[i[6]]+x[i[14]];
        r12=x[i[6]]-x[i[14]];
        r13=x[i[7]]+x[i[15]];
        r14=x[i[7]]-x[i[15]];
        r15=x[i[8]]+x[i[16]];
        r16=x[i[8]]-x[i[16]];
        t1 = r1 + r9;
        t2 = r1 - r9;
        t3 = r3 + r11;
        t4 = r3 - r11;
        t5 = r5 + r13;
        t6 = r5 - r13;
```

```
t7 = r7 + r15;
t8 = r7 - r15;
r1 = t1 + t5;
r3 = t1 - t5;
r5 = t3 + t7;
r7 = t3 - t7;
x[i[1]] = r1 + r5;
x[i[9]] = r1 - r5;
t1 = (t4 + t8) * c81;
t5 = (t4 - t8) * c81;
r9 = t2 + t5;
r11 = t2 - t5;
r13 = t6 + t1;
r15 = t6 - t1;
t1 = r4 + r16;
t2 = r4 - r16;
t3 = (r6 + r14) * c81;
t4 = (r6 - r14) * c81;
t5 = r8 + r12;
t6 = r8 - r12;
t7 = (t2 - t6) * c162;
t2 = c163 * t2 - t7;
t6 = c164 * t6 - t7;
t7 = r2 + t4;
t8 = r2 - t4;
r2 = t7 + t2;
r4 = t7 - t2;
r6 = t8 + t6;
r8 = t8 - t6;
t7 = (t1 + t5) * c165;
t2 = t7 - t1 * c164;
t4 = t7 - t5 * c163;
t6 = r10 + t3;
t8 = r10 - t3;
r10 = t6 + t2;
r12 = t6 - t2;
r14 = t8 + t4;
r16 = t8 - t4;
```

```
r1 =y[i[1]]+y[i[9]];
s2 =y[i[1]]-y[i[9]];
s3 =y[i[2]]+y[i[10]];
s4 =y[i[2]]-y[i[10]];
r5 =y[i[3]]+y[i[11]];
s6 =y[i[3]]  y[i[11]];
s7 =y[i[4]]+y[i[12]];
s8 =y[i[4]]-y[i[12]];
s9 =y[i[5]]+y[i[13]];
s10=y[i[5]]-y[i[13]];
s11=y[i[6]]+y[i[14]];
s12-y[i[6]]-y[i[14]];
s13=y[i[7]]+y[i[15]];
s14=y[i[7]]-y[i[15]];
s15=y[i[8]]+y[i[16]];
s16=y[i[8]]-y[i[16]];
t1 = r1 + s9;
t2 = r1 - s9;
t3 = s3 + s11;
t4 = s3 - s11;
t5 = r5 + s13;
t6 = r5 - s13;
t7 = s7 + s15;
t8 = s7 - s15;
r1 = t1 + t5;
s3 = t1 - t5;
r5 = t3 + t7;
s7 = t3 - t7;
y[i[ 1]] = r1 + r5;
y[i[ 9]] = r1 - r5;
x[i[ 5]] = r3 + s7;
x[i[13]] = r3 - s7;
y[i[ 5]] = s3 - r7;
y[i[13]] = s3 + r7;
t1 = (t4 + t8) * c81;
t5 = (t4 - t8) * c81;
s9 = t2 + t5;
```

```
s11 = t2 - t5;
s13 = t6 + t1;
s15 = t6 - t1;
t1 = s4 + s16;
t2 = s4 - s16;
t3 = (s6 + s14) * c81;
t4 = (s6 - s14) * c81;
t5 = s8 + s12;
t6 = s8 - s12;
t7 = (t2 - t6) * c162;
t2 = c163 * t2 - t7;
t6 = c164 * t6 - t7;
t7 = s2 + t4;
t8 = s2 - t4;
s2 = t7 + t2;
s4 = t7 - t2;
s6 = t8 + t6;
s8 = t8 - t6;
t7 = (t1 + t5) * c165;
t2 = t7 - t1 * c164;
t4 = t7 - t5 * c163;
t6 = s10 + t3;
t8 = s10 - t3;
s10 = t6 + t2;
s12 = t6 - t2;
s14 = t8 + t4;
s16 = t8 - t4;

x[i[ 2]] = r2 + s10;
x[i[16]] = r2 - s10;
x[i[ 3]] = r9 + s13;
x[i[15]] = r9 - s13;
x[i[ 4]] = r8 - s16;
x[i[14]] = r8 + s16;
x[i[ 6]] = r6 + s14;
x[i[12]] = r6 - s14;
x[i[ 7]] = r11- s15;
x[i[11]] = r11+ s15;
```

```
                        x[i[ 8]] = r4 - s12;
                        x[i[10]] = r4 + s12;
                        y[i[ 2]] = s2 - r10;
                        y[i[16]] = s2 + r10;
                        y[i[ 3]] = s9 - r13;
                        y[i[15]] = s9 + r13;
                        y[i[ 4]] = s8 + r16;
                        y[i[14]] = s8 - r16;
                        y[i[ 6]] = s6 - r14;
                        y[i[12]] = s6 + r14;
                        y[i[ 7]] = s11+ r15;
                        y[i[11]] = s11- r15;
                        y[i[ 8]] = s4 + r12;
                        y[i[10]] = s4 - r12;
                        break;
                      }
                  }
              }
         }
    a = malloc(n * sizeof(double));
    b = malloc(n * sizeof(double));
    unsc = 0;
    for (k=0;k<m;k++)
      unsc=unsc+n/ni[k];
    unsc = unsc % n;
    l = 0;
    for (k=0;k<n;k++)
       { a[k] = x[l];
         b[k] = y[l];
         l = l + unsc;
         if ( l > (n-1) ) l = l - n ;
       }
    for (k=0;k<n;k++)
       { x[k] = a[k];
         y[k] = b[k];
       }
    free(a);
    free(b);
```

```
}
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i) = Q^i \quad , \quad i = 0,1,\cdots,n-1$$

其离散傅立叶变换为

$$X(k) = \frac{1-Q^n}{1-QW^k} \quad , \quad k = 0,1,\cdots,n-1$$

这里 $W = e^{-j\frac{2\pi}{n}}$。选取 $Q=0.9+j\,0.3$，$n=45=5\times 9$，$m=2$，用素因子 FFT 算法计算离散傅立叶变换 $X(k)$。

主函数程序(文件名:pft.m)：

```
#include "math.h"
#include "pft.c"
main()
{ int i,j,n,m,ni[5];
  double a1,a2,x[45],y[45];
  n=45;
  m=2;
  ni[0]=5;
  ni[1]=9;
  a1 = 0.9;
  a2 = 0.3;
  x[0] = 1.0;
  y[0] = 0.0;
  for (i=1;i<n;i++)
    { x[i] = a1 * x[i-1] - a2 * y[i-1];
      y[i] = a2 * x[i-1] + a1 * y[i-1];
    }
  pft(x,y,n,m,ni);
  printf("\n DISCRETE FOURIER TRANSFORM\n");
  for (i=0;i<n/2;i++)
    { for (j=0;j<2;j++)
          printf(" %10.7f + J %10.7f",x[2*i+j],y[2*i+j]);
      printf("\n");
    }
  printf(" %11.7f + J %11.7f\n",x[n-1],y[n-1]);
}
```

运行结果：

离散傅立叶变换 $X(k)(k=0,1,\cdots,44)$ 为

1.2953730	+ J	3.0058661	2.4763272	+ J	5.0363348
13.1953398	+ J	8.5064906	4.2704752	+ J	−8.6202298
1.0911888	+ J	−4.2533418	0.6697366	+ J	−2.7314755
0.3571460	+ J	−1.9950949	0.5172978	+ J	−1.5612674
0.5015426	+ J	−1.2737836	0.4954450	+ J	−1.0678105
0.4937041	+ J	−0.9117504	0.4940833	+ J	−0.7883934
0.1955443	+ J	−0.6875687	0.4975721	+ J	−0.6028767
0.4998993	+ J	−0.5300837	0.5023842	+ J	−0.4662714
0.5049513	+ J	−0.4093561	0.5075622	+ J	−0.3578036
0.5101999	+ J	−0.3104517	0.5128600	+ J	−0.2663966
0.5155464	+ J	−0.2249168	0.5182683	+ J	−0.1854205
0.5210390	+ J	−0.1474081	0.5238755	+ J	−0.1104456
0.5267981	+ J	−0.0741421	0.5298309	+ J	−0.0381338
0.5330024	+ J	−0.0020681	0.5363465	+ J	0.0344097
0.5399041	+ J	0.0716705	0.5437254	+ J	0.1101173
0.5478722	+ J	0.1502029	0.5524231	+ J	0.1924535
0.5574789	+ J	0.2374986	0.5631718	+ J	0.2861126
0.5696785	+ J	0.3392738	0.5772418	+ J	0.3982506
0.5862035	+ J	0.4647333	0.5970602	+ J	0.5410392
0.6105602	+ J	0.6304493	0.6278819	+ J	0.7377841
0.6509801	+ J	0.8704402	0.6833236	+ J	1.0403943
0.7316197	+ J	1.2683983	0.8104348	+ J	1.5937721
0.9570999	+ J	2.1007988			

§1.10 Chirp Z-变换算法

一、功　能

在 Z 平面单位圆上计算有限长序列 $x(n)$ 的 Z 变换的采样值。

序列 $x(n)$ 的 DFT 实质上是其 Z-变换 $X(z)$ 在 Z 平面单位圆上等间隔采样点 $z_k=\frac{2\pi}{N}k$ 处的采样值。但实际应用中，往往只需要对信号的一小段频带进行高分辨率分析，这时通常使用 Chirp Z-变换算法。

二、方法简介

设 $x(n)$ 是长度为 N 的序列，其傅立叶变换为 $X(e^{j\omega})$。在单位圆任意一段弧上等间隔地取 M 个频率点

$$\omega_k = \omega_0 + k\Delta\omega \qquad k = 0,1,\cdots,M-1$$

式中 ω_0 为起始频率，$\Delta\omega$ 为频率增量。

在这些频率采样点上的傅立叶变换为

$$X(e^{j\omega_k}) = \sum_{n=0}^{N-1} x(n)e^{-j\omega_k n}$$

$$= \sum_{n=0}^{N-1} x(n)e^{-j\omega_0 n}e^{-j\Delta\omega kn} \quad , \quad k = 0,1,\cdots,M-1$$

设复常数 A 和 W 为

$$A = e^{j\omega_0}$$

$$W = e^{-j\Delta\omega}$$

则有

$$X(e^{j\omega_k}) = \sum_{n=0}^{N-1} x(n)A^{-n}W^{kn} \quad , \quad k = 0,1,\cdots,M-1$$

利用等式

$$nk = \frac{1}{2}[n^2 + k^2 - (k-n)^2]$$

可得

$$X(e^{j\omega_k}) = W^{\frac{k^2}{2}} \sum_{n=0}^{N-1} [x(n)A^{-n}W^{\frac{n^2}{2}}]W^{-\frac{(k-n)^2}{2}}$$

为清楚起见，令

$$\begin{cases} g(n) = x(n)A^{-n}W^{\frac{n^2}{2}} \\ h(n) = W^{-\frac{n^2}{2}} \end{cases}$$

从而得到卷积形式为

$$X(e^{j\omega_k}) = W^{\frac{k^2}{2}} \sum_{n=0}^{N-1} g(n)h(k-n) \quad , \quad k = 0,1,\cdots,M-1$$

如果 $\omega_0 = 0$，$M = N$，并且 $\Delta\omega = \frac{2\pi}{N}$，则此时的 Chirp Z-变换就是 DFT。

三、使用说明

1. 子函数语句

```
void czt(xr,xi,n,m,f1,f2)
```

2. 形参说明

xr ——双精度实型一维数组，其长度大于或等于(n+m−1)，且是 2 的整数次幂。开始存放输入数据的实部，最后存放变换结果的实部。

xi ——双精度实型一维数组，其长度大于或等于(n+m−1)，且是 2 的整数次幂。开始存放输入数据的虚部，最后存放变换结果的虚部。

n ——整型变量。输入数据的长度。

m ——整型变量。输出数据的长度,即频率采样点数。

f_1——双精度实型变量。起始数字频率,对应于公式中的$\frac{\omega_0}{2\pi}$,单位为 Hz-s。

f_2——双精度实型变量。终止数字频率,对应于公式中的$\frac{\omega_0+(M-1)\Delta\omega}{2\pi}$,单位为 Hz-s 。

四、子函数程序(文件名:czt.c)

```
#include "math.h"
#include "stdlib.h"
#include "fft.c"
void czt(xr,xi,n,m,f1,f2)
int m,n;
double f1,f2,xr[],xi[];
{ int i,j,n1,n2,len;
  double e,t,ar,ai,ph,pi,tr,ti, *wr, *wr1, *wi, *wi1;
  len=n+m-1;
  for (j=1,i=1;i<16;i++)
    { j=2*j;
      if (j>=len)
        { len=j;
          break;
        }
    }
  wr = malloc(len*sizeof(double));
  wi = malloc(len*sizeof(double));
  wr1 = malloc(len*sizeof(double));
  wi1 = malloc(len*sizeof(double));
  pi=3.14159265358979;
  ph=2.0*pi*(f2-f1)/(m-1);
  n1=(n>=m)? n:m;
  for (i=0;i<n1;i++)
    { e=ph*i*i/2.0;
      wr[i]=cos(e);
      wi[i]=sin(e);
      wr1[i]= wr[i];
      wi1[i]=-wi[i];
    }
  n2=len-n+1;
```

```
for (i=m;i<n2;i++)
  { wr[i]=0.0;
    wi[i]=0.0;
  }
for (i=n2;i<len;i++)
  { j=len-i;
    wr[i]=wr[j];
    wi[i]=wi[j];
  }
fft(wr,wi,len,1);
ph=-2.0*pi*f1;
for (i=0;i<n;i++)
  { e=ph*i;
    ar=cos(e);
    ai=sin(e);
    tr=ar*wr1[i] - ai*wi1[i];
    ti=ai*wr1[i] + ar*wi1[i];
    t = xr[i]*tr - xi[i]*ti;
    xi[i] = xr[i]*ti + xi[i]*tr;
    xr[i] = t;
  }
for (i=n;i<len;i++)
  { xr[i]=0.0;
    xi[i]=0.0;
  }
fft(xr,xi,len,1);
for (i=0;i<len;i++)
  { tr = xr[i]*wr[i] - xi[i]*wi[i];
    xi[i] = xr[i]*wi[i] + xi[i]*wr[i];
    xr[i] = tr;
  }
fft(xr,xi,len,-1);
for (i=0;i<m;i++)
  { tr = xr[i]*wr1[i] - xi[i]*wi1[i];
    xi[i] = xr[i]*wi1[i] + xi[i]*wr1[i];
    xr[i] = tr;
  }
free(wr);
```

```
  free(wi);
  free(wr1);
  free(wi1);
}
```

五、例　题

设输入序列为

$$x(i)=\begin{cases}0\ , & i=0,1,\cdots,9\\ \exp\left[-\frac{i-10}{15}\right]\sin\left[\frac{2\pi(i-10)}{16}\right], & i=10,11,\cdots,63\end{cases}$$

这是本章§1.6的例子，它的频谱有一个尖峰，我们用 Chirp Z-变换算法对尖峰附近的频谱进行高分辨率分析。选取 $n=64$，$m=200$，分析范围从 $f_1=0.0$ 到 $f_2=0.25$，然后画出其频谱图。

主函数程序(文件名:czt.m):

```
#include "stdio.h"
#include "math.h"
#include "czt.c"
main()
{ int i,n,m;
  double f1,f2,x,xr[512],xi[512];
  FILE *fp;
  n=64;
  for (i=0;i<10;i++)
    { xr[i]=0.;
      xi[i]=0.;
    }
  for (i=10;i<n;i++)
    { xr[i]=exp(-(i-10)/15.0)*sin(6.2831853*(i-10)/16.0);
      xi[i]=0.;
    }
  f1=0.0;
  f2=0.25;
  m=200;
  czt(xr,xi,n,m,f1,f2);
  printf("\n CHIRP Z-TRANSFORM\n");
  for (i=0;i<m;i+=2)
    { printf(" %10.7f + J %10.7f",xr[i],xi[i]);
      printf(" %10.7f + J %10.7f",xr[i+1],xi[i+1]);
```

```
        printf("\n");}
    for (i=0;i<m;i++)
      { xr[i]=sqrt(xr[i]*xr[i]+xi[i]*xi[i]);}
    fp=fopen("czt.dat","w");
    for (i=0;i<m;i++)
      { x=f1+i*(f2-f1)/(m-1);
        fprintf(fp,"%f   %f\n",x,xr[i]);
      }
    close(fp);
  }
```

运行结果：

Chirp Z-变换的前 20 点数据为：

2.4716438	− J	−0.0000000	2.4593515	+ J	−0.2242989
2.4236981	− J	−0.4431982	2.3680238	+ J	−0.6525196
2.2968822	+ J	−0.8501945	2.2148079	+ J	−1.0365765
2.1250899	+ J	−1.2140878	2.0288847	+ J	−1.3862879
1.9249178	+ J	−1.5566124	1.8098659	+ J	−1.7271207
1.6793298	+ J	−1.8975963	1.5291442	+ J	−2.0652549
1.3566765	+ J	−2.2251580	1.1617561	+ J	−2.3712392
0.9469668	+ J	−2.4976831	0.7171978	+ J	−2.6002902
0.4785472	+ J	−2.6774518	0.2368496	+ J	−2.7304486
−0.0037841	+ J	−2.7629584	−0.2420132	+ J	−2.7798645

输入序列 $x(i)$的频谱如图 2—1—2 所示。

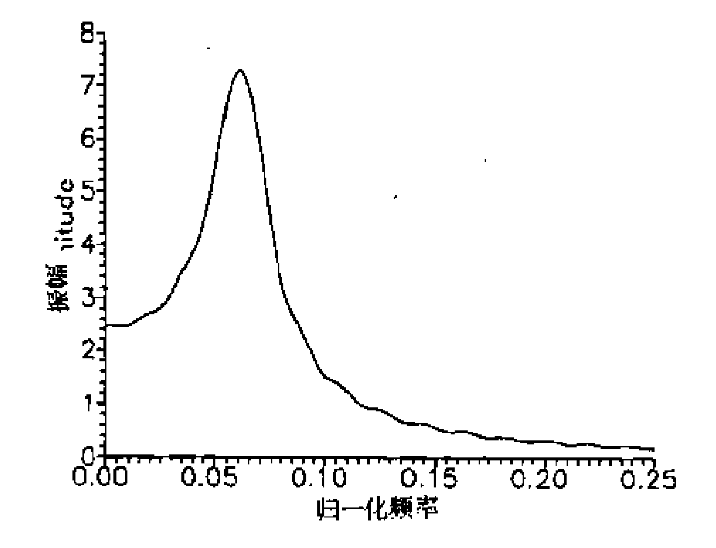


图 2—1—2　序列 $x(i)$的频谱

第二章　快速离散正交变换

§2.1　快速哈特莱(Hartley)变换

一、功　能

计算快速哈特莱(Hartley)变换。

二、方法简介

实序列 $x(n)$ 的离散哈特莱变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)\mathrm{cas}(\frac{2\pi}{N}nk),\quad k=0,1,\cdots,N-1$$

将序列 $x(n)$ 按序号 n 的奇偶分成两组，即

$$\begin{matrix} x_1(n)=x(2n) \\ x_2(n)=x(2n+1) \end{matrix}\quad n=0,1,\cdots,\frac{N}{2}-1$$

因此，$x(n)$ 的离散哈特莱变换可表示为

$$X(k)=\sum_{n=0}^{N/2-1}x_1(n)\mathrm{cas}(\frac{2\pi}{N}2nk)+\sum_{n=0}^{N/2-1}x_2(n)\mathrm{cas}(\frac{2\pi}{N}(2n+1)k)$$

设 $x_1(n)$ 和 $x_2(n)$ 的 $N/2$ 点离散哈特莱变换分别为 $X_1(k)$ 和 $X_2(k)$，并记 $c(k)=\cos\frac{2\pi}{N}k$，$s(k)=\sin\frac{2\pi}{N}k$，我们得到

$$\begin{cases} X(k)=X_1(k)+\left[c(k)X_2(k)+s(k)X_2(\frac{N}{2}-k)\right] \\ X(\frac{N}{2}+k)=X_1(k)-\left[c(k)X_2(k)+s(k)X_2(\frac{N}{2}-k)\right] \\ X(\frac{N}{2}-k)=X_1(\frac{N}{2}-k)+\left[s(k)X_2(k)-c(k)X_2(\frac{N}{2}-k)\right] \\ X(N-k)=X_1(\frac{N}{2}-k)-\left[s(k)X_2(k)-c(k)X_2(\frac{N}{2}-k)\right] \end{cases}$$

$$k=0,1,\cdots,\frac{N}{4}-1$$

通过上面的推导可以看出，N 点的离散哈特莱变换可以分解为两个 $N/2$ 点的离散哈特莱变换，每个 $N/2$ 点的离散哈特莱变换又可以分解为两个 $N/4$ 点的离散哈特莱变换。依此类推，当 N 为 2 的整数次幂时($N=2^M$)，由于每分解一次降低一阶幂次，所以通过 M 次的分解，最后全部成为一系列 2 点离散哈特莱变换运算。以上就是按时间抽取的快速

离散哈特莱变换(FHT)算法。

离散哈特莱反变换为

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)\text{cas}(\frac{2\pi}{N}nk),\quad n=0,1,\cdots,N-1$$

显然，除了常数因子 $1/N$ 之外，它与离散哈特莱正变换具有完全相同的结构。因此，离散哈特莱反变换与离散哈特莱正变换可以共用一个程序。

三、使用说明

1. 子函数语句

void fht(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:fht.c)

```
#include "math.h"
void fht(x,n)
int n;
double x[];
{ int i,j,k,m,l1,l2,l3,l4,n1,n2,n4;
  double a,e,c,s,t,t1,t2;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2 * j;
      if (j==n) break;
    }
  n1 = n-1;
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { t = x[j];
          x[j] = x[i];
          x[i] = t;
        }
      k = n/2;
      while ( k < (j+1) )
        { j = j-k;
          k = k/2;
```

```
            }
        j = j+k;
    }
for (i=0;i<n;i+=2)
    { t = x[i];
      x[i] = t+x[i+1];
      x[i+1] = t-x[i+1];
    }
n2=1;
for (k=2;k<=m;k++)
    { n4 = n2;
      n2 = n4+n4;
      n1 = n2+n2;
      e = 6.283185307179586/n1;
      for (j=0;j<n;j+=n1)
        { l2=j+n2;
          l3=j+n4;
          l4=l2+n4;
          t=x[j];
          x[j]=t+x[l2];
          x[l2]=t-x[l2];
          t=x[l3];
          x[l3]=t+x[l4];
          x[l4]=t-x[l4];
          a=e;
          for (i=1;i<n4;i++)
            { l1=j+i;
              l2=j-i+n2;
              l3=l1+n2;
              l4=l2+n2;
              c=cos(a);
              s=sin(a);
              t1=x[l3]*c+x[l4]*s;
              t2=x[l3]*s-x[l4]*c;
              a=(i+1)*e;
              t=x[l1];
              x[l1]=t+t1;
              x[l3]=t-t1;
```

```
                    t=x[l2];
                    x[l2]=t+t2;
                    x[l4]=t-t2;
                  }
              }
          }
      }
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i)=\begin{cases}0, & i=0,1,\cdots,9\\ \exp\left[-\dfrac{i-10}{15}\right]\sin\left[\dfrac{2\pi(i-10)}{16}\right], & i=10,11,\cdots,31\end{cases}$$

选取序列长度 $n=32$，计算其离散哈特莱变换。

主函数程序(文件名:fht.m):

```
#include "math.h"
#include "fht.c"
main()
{ int i,n;
  double x[32];
  n=32;
  for (i=0;i<10;i++)
    { x[i]=0.0; }
  for (i=10;i<n;i++)
    { x[i]=exp(-(i-10)/15.0) * sin(6.2831853 * (i-10)/16.0); }
  fht(x,n);
  printf("\n DISCRETE HARTLEY TRANSFORM\n");
  for (i=0;i<n;i+=4)
    { printf("      %10.7f      %10.7f",x[i],x[i+1]);
      printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
}
```

运行结果:

序列 $x(i)$ 的离散哈特莱变换 $X(k)$ 为

```
     2.6897298      1.4113815      -0.9471247      -0.8476419
    -1.5230888      0.2704962      -0.2260261      -0.5911014
```

−0.0159660	−0.0789627	−0.3726402	−0.0878206
−0.0183085	−0.2688064	−0.1200390	0.0224531
−0.1999339	−0.1459628	0.0615104	−0.1389357
−0.1830083	0.1098251	−0.0629875	−0.2619540
0.1849758	0.0835138	−0.5129741	0.3621105
0.6329974	−2.9488820	8.1966895	−4.4735182

§2.2 基4快速哈特莱(Hartley)变换

一、功 能

用基4快速算法计算离散哈特莱(Hartley)变换。

二、方法简介

实序列 $x(n)$ 的离散哈特莱变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)\cos(\frac{2\pi}{N}nk),\quad k=0,1,\cdots,N-1$$

对 $x(n)$ 进行抽取,即把 $x(n)$ 分解为

$$x(n)=x_1(n)+x_2(n)+x_3(n)+x_4(n)$$

式中

$$x_1(n)=x(4n),\quad x_2(n)=x(4n+1),\quad x_3(n)=x(4n+2),\quad x_4(n)=x(4n+3)$$

$$n=0,1,\cdots,\frac{N}{4}-1$$

设 $x_1(n)$、$x_2(n)$、$x_3(n)$、$x_4(n)$ 的离散哈特莱变换分别为 $X_1(k)$、$X_2(k)$、$X_3(k)$、$X_4(k)$,那么 $X(k)$ 可以分解为

$$\begin{aligned}X(k)=&X_1(k)+\cos(\frac{2\pi}{N}k)X_2(k)+\sin(\frac{2\pi}{N}k)X_2(N-k)\\&+\cos(\frac{2\pi}{N}2k)X_3(k)+\sin(\frac{2\pi}{N}2k)X_3(N-k)\\&+\cos(\frac{2\pi}{N}3k)X_4(k)+\sin(\frac{2\pi}{N}3k)X_4(N-k)\end{aligned}$$

$$k=0,1,\cdots,\frac{N}{4}-1$$

这样,就将一个 N 点的离散哈特莱变换转化为四个 $N/4$ 点离散哈特莱变换来计算。依此类推,直至分解到最后一级。以上就是按时间抽取的基4快速哈特莱变换算法。

三、使用说明

1. 子函数语句

```
void fht4(x,n)
```

2. **形参说明**

x ——双精度实型一维数组，长度为n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是4的整数次幂，即 $n=4^m$。

四、子函数程序(文件名:fht4.c)

```
#include "math.h"
void fht4(x,n)
int n;
double x[];
{ int i,j,k,m,n1,n2,n4,l11,l12,l13,l14,l21,l22,l23,l24;
  double a,e,c1,c2,c3,s1,s2,s3,t1,t2,t3,t4,t12,t13,t14,t22,t23,t24;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 4 * j;
      if (j==n) break;
    }
  n1 = n-1;
  for (j=0,i=0;i<n1;i++)
    { if ( i < j )
        { a = x[j];
          x[j] = x[i];
          x[i] = a;
        }
      k = n/4;
      while ( 3 * k < (j+1) )
        { j = j - 3 * k;
          k = k/4;
        }
      j = j + k;
    }
  for (i=0;i<n;i+=4)
    { t1 = x[i]+x[i+1];
      t2 = x[i]-x[i+1];
      t3 = x[i+2]+x[i+3];
      t4 = x[i+2]-x[i-3];
      x[i] = t1+t3;
      x[i+1] = t1-t3;
```

```
        x[i+2] = t2+t4;
        x[i+3] = t2-t4;
      }
    n2 = 1;
    for (k=1;k<m;k++)
      { n4 = 2 * n2;
        n2 = 4 * n2;
        n1 = 4 * n2;
        e = 6.283185307179586/n1;
        for (i=0;i<n;i+=n1)
          { l12 = i + n2;
            l13 = l12 + n2;
            l14 = l13 + n2;
            t1 = x[i] + x[l12];
            t2 = x[i] - x[l12];
            t3 = x[l13] + x[l14];
            t4 = x[l13] - x[l14];
            x[i] = t1 + t3;
            x[l12] = t1 - t3;
            x[l13] = t2 + t4;
            x[l14] = t2 - t4;
            l21 = i + n4;
            l22 = l21 + n2;
            l23 = l22 + n2;
            l24 = l23 + n2;
            t1 = x[l21];
            t2 = x[l22] * sqrt(2.0);
            t3 = x[l23];
            t4 = x[l24] * sqrt(2.0);
            x[l21] = t1 + t2 + t3;
            x[l22] = t1 - t3 + t4;
            x[l23] = t1 - t2 + t3;
            x[l24] = t1 - t3 - t4;
            a = e;
            for (j=1;j<n4;j++)
              { l11 = i + j;
                l12 = l11 + n2;
                l13 = l12 + n2;
```

```
                l14 = l13 + n2;
                l21 = i + n2 - j;
                l22 = l21 + n2;
                l23 = l22 + n2;
                l24 = l23 + n2;
                c1 = cos(a);
                s1 = sin(a);
                c2 = cos(2*a);
                s2 = sin(2*a);
                c3 = cos(3*a);
                s3 = sin(3*a);
                a = (j+1) * e;
                t12 = x[l12]*c1+x[l22]*s1;
                t13 = x[l13]*c2+x[l23]*s2;
                t14 = x[l14]*c3+x[l24]*s3;
                t22 = x[l12]*s1-x[l22]*c1;
                t23 = x[l13]*s2-x[l23]*c2;
                t24 = x[l14]*s3-x[l24]*c3;
                t1 = x[l21]+t23;
                t2 = x[l21]-t23;
                t3 = t22+t24;
                t4 = t12-t14;
                x[l24] = t2-t3;
                x[l23] = t1-t4;
                x[l22] = t2+t3;
                x[l21] = t1+t4;
                t1 = x[l11]+t13;
                t2 = x[l11]-t13;
                t3 = t24-t22;
                t4 = t12+t14;
                x[l14] = t2-t3;
                x[l13] = t1-t4;
                x[l12] = t2+t3;
                x[l11] = t1+t4;
              }
          }
      }
  }
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i) = e^{-0.5i} \quad , \quad i = 0,1,\cdots,n-1$$

选取序列长度 $n=16$,用基 4 快速算法计算 $x(i)$ 的离散哈特莱变换。

主函数程序(文件名:fht4.m):

```
#include "math.h"
#include "fht4.c"
main()
{ int i,n;
  double x[16];
  n=16;
  for (i=0;i<n;i++)
    { x[i]=exp(-0.5*i); }
  fht4(x,n);
  printf("\n DISCRETE HARTLEY TRANSFORM\n");
  for (i=0;i<n;i+=4)
    { printf("      %10.7f      %10.7f",x[i],x[i+1]);
      printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
}
```

运行结果:

序列 $x(i)$ 的离散哈特莱变换 $X(k)$ 为

```
2.5406415     2.7169890     1.9596825     1.4693635
1.1740740     0.9780423     0.8344287     0.7200304
0.6222505     0.5335552     0.4491576     0.3665321
0.2875526     0.2295771     0.2787376     0.8393853
```

§2.3　分裂基快速哈特莱(Hartley)变换

一、功　能

用分裂基快速算法计算离散哈特莱(Hartley)变换。

二、方法简介

实序列 $x(n)$ 的离散哈特莱变换为

$$X(k)=\sum_{n=0}^{N-1}x(n)\cos(\frac{2\pi}{N}nk),\quad k=0,1,\cdots,N-1$$

将 $X(k)$按序号 k 的奇偶分成两组。当 k 为偶数时，进行基 2 频率抽取分解，$X(k)$可表示为

$$X(2k)=\sum_{n=0}^{N/2-1}[x(n)+x(n+\frac{N}{2})]\cos(\frac{2\pi}{N}2nk)\quad,\quad k=0,1,\cdots,\frac{N}{2}-1$$

当 k 为奇数时，进行基 4 频率抽取分解，$X(k)$可表示为

$$X(4k+1)=\sum_{n=0}^{N/4-1}\Big\{\Big[x(n)-x(n+\frac{N}{2})+x(\frac{N}{4}-n)-x(\frac{3N}{4}-n)\Big]\cos(\frac{2\pi}{N}n)$$

$$+\Big[x(n+\frac{3N}{4})-x(n+\frac{N}{4})+x(\frac{N}{2}-n)-x(N-n)\Big]$$

$$\sin(\frac{2\pi}{N}n)\Big\}\cos(\frac{2\pi}{N}4kn)$$

$$X(4k+3)=\sum_{n=0}^{N/4-1}\Big\{\Big[x(n)-x(n+\frac{N}{2})-x(\frac{N}{4}-n)+x(\frac{3N}{4}-n)\Big]\cos(\frac{2\pi}{N}3n)$$

$$+\Big[x(n+\frac{3N}{4})-x(n+\frac{N}{4})-x(\frac{N}{2}-n)+x(N-n)\Big]$$

$$\sin(\frac{2\pi}{N}3n)\Big\}\cos(\frac{2\pi}{N}4kn)$$

$$k=0,1,\cdots,\frac{N}{4}-1$$

这样，一个 N 点的离散哈特莱变换就分解为一个 $N/2$ 点的离散哈特莱变换和两个 $N/4$ 点的离散哈特莱变换。上面的 $N/2$ 点离散哈特莱变换又可分解为一个 $N/4$ 点的离散哈特莱变换和两个 $N/8$ 点的离散哈特莱变换，而两个 $N/4$ 点的离散哈特莱变换也分别可以分解为一个 $N/8$ 点的离散哈特莱变换和两个 $N/16$ 点的离散哈特莱变换。依此类推，直至分解到最后一级为止。这就是按频率抽取的分裂基快速哈特莱变换算法。

三、使用说明

1. 子函数语句

void srfht(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:srfht.c)

```
#include "math.h"
void srfht(x,n)
int n;
double x[];
```

```
{ int i,j,k,m,n1,n2,n4,n8,is,id;
  int l11,l12,l13,l14,l21,l22,l23,l24;
  double a,a3,e,c1,c3,s1,s3,st,t1,t2,t3,t4,t5;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2*j;
      if (j==n) break;
    }
  st = sqrt(2.0);
  n2 = 2*n;
  for (k=1;k<m;k++)
    { is = 0;
      id = n2;
      n2 = n2/2;
      n4 = n2/4;
      n8 = n2/8;
      e = 6.283185307179586/n2;
      do
        { for (i=is;i<n;i+=id)
            { l12 = i + n4;
              l13 = l12 + n4;
              l14 = l13 + n4;
              t1 = x[l12] - x[l14];
              t2 = x[i] + x[l13];
              t3 = x[l12] + x[l14];
              t4 = x[i] - x[l13];
              x[l14] = t4 - t1;
              x[l13] = t4 + t1;
              x[l12] = t3;
              x[i] = t2;
              if (n4==1) continue;
              l21 = i + n8;
              l22 = l21 + n4;
              l23 = l22 + n4;
              l24 = l23 + n4;
              t1 = x[l22];
              t2 = x[l23];
              t3 = x[l24];
```

```
x[l24] = (t1 - t3) * st;
x[l23] = (x[l21] - t2) * st;
x[l22] = t1 + t3;
x[l21] = x[l21] + t2;
a = e;
for (j=1;j<n8;j++)
  { l11 = i + j;
    l12 = l11 + n4;
    l13 = l12 + n4;
    l14 = l13 + n4;
    l21 = i + n4 - j;
    l22 = l21 + n4;
    l23 = l22 + n4;
    l24 = l23 + n4;
    a3 = 3*a;
    c1 = cos(a);
    s1 = sin(a);
    c3 = cos(a3);
    s3 = sin(a3);
    a = (j+1) * e;
    t5 = x[l21]-x[l23];
    t2 = x[l11]-x[l13];
    t1 = t2+t5;
    t2 = t2-t5;
    t5 = x[l22]-x[l24];
    t4 = x[l14]-x[l12];
    t3 = t4+t5;
    t4 = t4-t5;
    x[l11] = x[l11]+x[l13];
    x[l12] = x[l12]+x[l14];
    x[l21] = x[l21]+x[l23];
    x[l22] = x[l22]+x[l24];
    x[l13] = t1*c1+t3*s1;
    x[l14] = t2*c3-t4*s3;
    x[l23] = t1*s1-t3*c1;
    x[l24] = t2*s3+t4*c3;
  }
}
```

```
                is = 2 * id - n2;
                id = 4 * id;
            } while ( is < (n-2) );
        }
    is = 0;
    id = 4;
    do
        { for(i=is;i<n;i+=id)
            { t1 = x[i];
              x[i] = t1 - x[i+1];
              x[i+1]= t1 - x[i+1];
            }
          is = 2 * id-2;
          id = 4 * id;
        } while ( is < (n-1) );

    n1 = n-1;
    for (j=0,i=0;i<n1;i++)
        { if ( i < j )
            { t1 = x[j];
              x[j] = x[i];
              x[i] = t1;
            }
          k = n/2;
          while ( k < (j+1) )
            { j = j-k;
              k = k/2;
            }
          j = j+k;
        }
}
```

五、例　题

设输入序列 $x(i)$ 为

$$x(i)=\begin{cases}0, & i=0,1,\cdots,9\\ \exp\left[-\dfrac{i-10}{15}\right]\sin\left[\dfrac{2\pi(i-10)}{16}\right], & i=10,11,\cdots,31\end{cases}$$

选取序列长度 $n=32$，用分裂基快速算法计算其离散哈特莱变换。

主函数程序(文件名:srfht.m):

```
#include "math.h"
#include "srfht.c"
main()
{ int i,n;
  double x[32];
  n=32;
  for (i=0;i<10;i++)
    { x[i]=0.0; }
  for (i=10;i<n;i++)
    { x[i]=exp(-(i-10)/15.0)*sin(6.2831853*(i-10)/16.0); }
  srfht(x,n);
  printf("\n DISCRETE HARTLEY TRANSFORM\n");
  for (i=0;i<n;i+=4)
    { printf("        %10.7f        %10.7f",x[i],x[i+1]);
      printf("        %10.7f        %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
}
```

运行结果:

序列 $x(i)$ 的离散哈特莱变换 $X(k)$ 为

2.6897298	1.4113815	−0.9471247	−0.8476419
−1.5230888	0.2704962	−0.2260261	−0.5911014
−0.0159660	−0.0789627	−0.3726402	−0.0878206
−0.0183085	−0.2688064	−0.1200390	0.0224531
−0.1999339	−0.1459628	0.0615104	−0.1389357
−0.1830083	0.1098251	−0.0629875	−0.2619540
0.1849758	0.0835138	−0.5129741	0.3621105
0.6329974	−2.9488820	8.1966895	−4.4735182

§2.4 快速离散余弦变换

一、功 能

用 N 点快速傅立叶反变换计算 N 点离散余弦变换。

二、方法简介

实序列 $x(n)$ 的离散余弦变换为

$$X(k)=\sqrt{\frac{2}{N}}c(k)\sum_{n=0}^{N-1}x(n)\cos\left[\frac{(2n+1)k\pi}{2N}\right]\quad,\quad k=0,1,\cdots,N-1$$

其中

$$c(k)=\begin{cases}1/\sqrt{2}\,, & k=0\\ 1\quad, & k=1,2,\cdots,N-1\end{cases}$$

设

$$\overline{X}(k)=\sum_{n=0}^{N-1}x(n)\cos\left[\frac{(2n+1)k\pi}{2N}\right],\quad k=0,1,\cdots,N-1$$

显然

$$X(k)=\sqrt{\frac{2}{N}}c(k)\overline{X}(k)$$

为了用 FFT 计算离散余弦变换，我们定义一个新序列 y(n)

$$\begin{cases}y(n)=x(2n) & ,\quad n=0,1,\cdots,N/2-1\\ y(N-1-n)=x(2n+1) & ,\quad n=0,1,\cdots,N/2-1\end{cases}$$

这时有

$$\begin{aligned}\overline{X}(k)=&\sum_{n=0}^{N/2-1}y(n)\cos\left[\frac{(4n+1)k\pi}{2N}\right]\\&+\sum_{n=0}^{N/2-1}y(N-1-n)\cos\left[\frac{(4n+3)k\pi}{2N}\right]\\=&\sum_{n=0}^{N-1}y(n)\cos\left[\frac{(4n+1)k\pi}{2N}\right]\quad,\quad k=0,1,\cdots,N-1\end{aligned}$$

设 $y(n)$ 的离散傅立叶反变换 $Y_F(k)$ 为

$$Y_F(k)=\mathrm{IDFT}[y(n)]=\frac{1}{N}\sum_{n=0}^{N-1}y(n)e^{j\frac{2\pi nk}{N}}\quad,\quad k=0,1,\cdots,N-1$$

并令

$$H(k)=Ne^{j\pi k/2N}Y_F(k)\quad,\quad k=0,1,\cdots,N-1$$

从而 $x(n)$ 的离散余弦变换 $X(k)$ 可表示为

$$\begin{cases}X(k)=\sqrt{\frac{2}{N}}c(k)\mathrm{Re}[H(k)] & ,\quad k=0,1,\cdots,\frac{N}{2}\\ X(N-k)=\sqrt{\frac{2}{N}}c(k)\mathrm{Im}[H(k)] & ,\quad k=0,1,\cdots,\frac{N}{2}\end{cases}$$

由此可见，实序列 $x(n)$ 的 N 点离散余弦变换 $X(k)$ 可以用一个 N 点序列 $y(n)$ 的离散傅立叶反变换来计算。注意到序列 $y(n)$ 是实数，因此可以采用实序列快速傅立叶变换算法，以进一步减少运算量。

三、使用说明

1. 子函数语句

void fct(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:fct.c)

```
#include "stdlib.h"
#include "math.h"
#include "rfft.c"
void fct(x,n)
int n;
double x[];
{ int i,n1;
  double q,c,s, *y;
  y = malloc(n * sizeof(double));
  n1 = n/2;
  for (i=0;i<n1;i++)
    { y[i] = x[2*i];
      y[n-1-i] = x[2*i+1];
    }
  rfft(y,n);
  q = 4.0*atan(1.)/(2*n);
  x[0] = y[0];
  x[n1] = sin(n1*q)*y[n1];
  for (i=1;i<n1;i++)
    { c = cos(i*q);
      s = sin(i*q);
      x[i] = c*y[i]+s*y[n-i];
      x[n-i] = s*y[i]-c*y[n-i];
    }
  c = 1.0/sqrt((double)n);
  x[0] = c*x[0];
  c = sqrt(2.0/n);
  for (i=1;i<n;i++)
```

```
    { x[i] = c * x[i]; }
  free(y);
}
```

五、例 题

设 $n=16$,计算序列 $x(i)=e^{-0.5i}(i=0,1,\cdots,n-1)$的离散余弦变换。

主函数程序(文件名:fct.m):

```
#include "math.h"
#include "fct.c"
main()
{ int i,n;
  double x[16];
  n=16;
  for (i=0;i<n;i++)
    x[i]=exp(-0.5*i);
  fct(x,n);
  printf("\n Discrete Cosine Transform\n");
  for (i=0;i<n;i+=4)
    { printf("      %10.7f      %10.7f",x[i],x[i+1]);
      printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
}
```

运行结果:

序列 $x(i)$的离散余弦变换为

0.6351604	0.7774744	0.5518504	0.3706744
0.2518645	0.1768562	0.1279561	0.0950930
0.0718882	0.0550200	0.0421707	0.0321278
0.0239114	0.0169980	0.0109019	0.0053330

§ 2.5 快速离散余弦反变换

一、功 能

用 N 点快速傅立叶反变换计算 N 点离散余弦反变换。

二、方法简介

实序列 $X(k)$的离散余弦反变换为

$$x(n)=\sqrt{\frac{2}{N}}\sum_{k=0}^{N-1} c(k)X(k)\cos\left[\frac{(2n+1)k\pi}{2N}\right] \quad , \quad n=0,1,\cdots,N-1$$

其中

$$c(k)=\begin{cases}1/\sqrt{2} & , \quad k=0 \\ 1 & , \quad k=1,2,\cdots,N-1\end{cases}$$

显然,$x(\cdot)$的偶数项可表示为

$$x(2n)=\mathrm{Re}\left\{\sum_{k=0}^{N-1}\left[\sqrt{\frac{2}{N}}c(k)X(k)e^{j2\pi nk/N}\right]e^{jk\pi/Nnk}\right\} \quad , \quad n=0,1,\cdots,N-1$$

为了计算 $x(\cdot)$的奇数项,注意到

$$x(2n+1)=x[2(N-1-n)]$$

因此,通过计算序列 $\sqrt{\frac{2}{N}}c(k)X(k)e^{jk\pi/2N}$ 的 IDFT 的实部,可以得到离散余弦反变换 $x(n)$。计算步骤如下:

(1) 利用实序列 $X(k)$形成复序列 $Y(k)$

$$Y(k)=\sqrt{\frac{2}{N}}c(k)X(k)e^{jk\pi/2N}$$

(2) 计算复序列 $Y(k)$的离散傅立叶反变换 $y(n)$;

(3) 求 $y(n)$的实部: $z(n)=N\,\mathrm{Re}[y(n)]$;

(4) 计算下式,从而得到离散余弦反变换 $x(n)$。

$$\begin{cases}x(2n)=z(n) & , \quad n=0,1,\cdots,\frac{N}{2}-1 \\ x(2n+1)=z(N-1-n) & , \quad n=0,1,\cdots,\frac{N}{2}-1\end{cases}$$

三、使用说明

1. 子函数语句

void ifct(x,n)

2. 形参说明

x ——双精度实型一维数组,长度为 n。开始时存放要变换的实数据,最后存放变换结果。

n ——整型变量。数据长度,必须是 2 的整数次幂,即 $n=2^m$。

四、子函数程序(文件名:ifct.c)

```
#include "math.h"
#include "stdlib.h"
#include "fft.c"
void ifct(x,n)
int n;
```

```
double x[];
{ int i,n1;
  double q,c, * y;
  y = malloc(n * sizeof(double));
  n1 = n/2;
  q = 4. * atan(1.0)/(2 * n);
  x[0] /= sqrt(2.0);
  for (i=0;i<n;i++)
    { y[i] = x[i] * sin(i * q);
      x[i] = x[i] * cos(i * q);
    }
  fft(x,y,n,-1);
  for (i=0;i<n;i++)
    y[i] = n * x[i];
  c = sqrt(2.0/n);
  for (i=0;i<n1;i++)
    { x[2 * i] = c * y[i];
      x[2 * i+1] = c * y[n-1-i];
    }
  free(y);
}
```

五、例　题

设 $n=16$，计算序列 $x(i)=e^{-0.5i}$ $(i=0,1,\cdots,n-1)$ 的离散余弦变换和离散余弦反变换。

主函数程序（文件名：ifct.m）：

```
#include "math.h"
#include "fct.c"
#include "ifct.c"
main()
{ int i,n;
  double x[16];
  n=16;
  for (i=0;i<n;i++)
    x[i]=exp(-0.5 * i);
  printf("\n Input Sequence\n");
  for (i=0;i<n;i+=4)
    { printf("        %10.7f        %10.7f",x[i],x[i+1]);
```

```
        printf("        %10:7f        %10.7f\n",x[i+2],x[i+3]);
      }
    fct(x,n);
    printf("\n Discrete Cosine Transform\n");
    for (i=0;i<n;i+=4)
      { printf("        %10.7f        %10.7f",x[i],x[i+1]);
        printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
      }
    ifct(x,n);
    printf("\n Inverse Discrete Cosine Transform\n");
    for (i=0;i<n;i+=4)
      { printf("        %10.7f        %10.7f",x[i],x[i+1]);
        printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
      }
  }
```

运行结果：

输入序列 $x(i)$

```
1.0000000      0.6065307      0.3678794      0.2231302
0.1353353      0.0820850      0.0497871      0.0301974
0.0183156      0.0111090      0.0067379      0.0040868
0.0024788      0.0015034      0.0009119      0.0005531
```

序列 $x(i)$的离散余弦变换 $X(k)$

```
0.6351604      0.7774744      0.5518504      0.3706744
0.2518645      0.1768562      0.1279561      0.0950930
0.0718882      0.0550200      0.0421707      0.0321278
0.0239114      0.0169980      0.0109019      0.0053330
```

离散余弦反变换 $x(i)$

```
1.0000000      0.6065307      0.3678794      0.2231302
0.1353353      0.0820850      0.0497871      0.0301974
0.0183156      0.0111090      0.0067379      0.0040868
0.0024788      0.0015034      0.0009119      0.0005531
```

§2.6　$N=8$ 点快速离散余弦变换

一、功　能

用 B. G. Lee 算法计算 $N=8$ 点离散余弦变换。

二、方法简介

实序列 $x(n)$ 的离散余弦变换为

$$X(k)=\sqrt{\frac{2}{N}}c(k)\sum_{n=0}^{N-1}x(n)\cos\left[\frac{(2n+1)k\pi}{2N}\right]\quad,\quad k=0,1,\cdots,N-1$$

其中

$$c(k)=\begin{cases}1/\sqrt{2} & , \quad k=0\\ 1 & , \quad k=1,2,\cdots,N-1\end{cases}$$

记

$$X(k)=\sqrt{\frac{2}{N}}c(k)\overline{X}(k)\quad,\quad k=0,1,\cdots,N-1$$

离散余弦变换的快速算法如下：

(1) 计算

$$\begin{cases}g(n)=x(n)+x(N-1-n)\\ h(n)=[x(n)-x(N-1-n)]/2\cos\dfrac{(2n+1)\pi}{2N}\end{cases}$$

$$n=0,1,\cdots,N/2-1$$

(2) 计算两个 $N/2$ 点的离散余弦变换

$$\begin{cases}G(k)=\displaystyle\sum_{n=0}^{N/2-1}g(n)\cos\frac{(2n+1)k\pi}{N}\\ H(k)=\displaystyle\sum_{n=0}^{N/2-1}h(n)\cos\frac{(2n+1)k\pi}{N}\end{cases}$$

$$k=0,1,\cdots,N/2-1$$

(3) 蝶形运算

$$\begin{cases}\overline{X}(2k)=G(k)\\ \overline{X}(2k+1)=H(k+1)+H(k)\end{cases}$$

$$k=0,1,\cdots,N/2-1$$

这样就把一个 N 点 DCT 分解为两个 $N/2$ 点的 DCT。依此类推，直至分解到最后一级为止。

三、使用说明

1. 子函数语句

```
void fct8(x,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:fct8.c)

```
#include "math.h"
void fct8(x,n)
int n;
double x[];
{ int i,j,k,n1,n2;
  double t,pi,coef[8];
  n1 = n/2;
  n2 = n/4;
  pi = 3.141592654;
  t = x[2]; x[2] = x[3]; x[3] = t;
  t = x[4]; x[4] = x[7]; x[7] = x[5]; x[5] = x[6]; x[6] = t;
  for (i=0;i<n1;i++)
    { t = x[i] + x[n1+i];
      x[i+n1] = x[i] - x[n1+i];
      x[i] = t;
    }
  for(i=0;i<n2;i++)
    { coef[i] = 1/(2*cos((2*i+1)*pi/(2*n)));
      coef[i+n1] = 1/(2*cos((2*i+1)*2*pi/(2*n)));
      coef[i+n2] = 1/(2*cos((n-(2*i+1))*pi/(2*n)));
    }
  for (i=0;i<n2;i++)
    { x[i+n1] = x[i+n1] * coef[i];
      x[i+n1+n2] = x[i+n1+n2] * coef[i+n2];
    }
  for (i=0;i<n2;i++)
  for (j=0;j<n2;j++)
    { t = x[4*i+j] + x[4*i+j+n2];
      x[4*i+j+n2] = x[4*i+j] - x[4*i+j+n2];
      x[4*i+j] = t;
    }
  for (i=0;i<n2;i++)
  for (j=0;j<n2;j++)
    { x[4*i+j+n2] = x[4*i+j+n2]*coef[j+n1]; }
  for (i=0;i<n1;i++)
    { t = x[2*i] + x[2*i+1];
```

```
        x[2*i+1] = x[2*i] - x[2*i+1];
        x[2*i] = t;
      }
    t = 1/(2*cos((4*pi/(2*n))));
    for (i=1;i<n;i+=2)
      { x[i] = x[i] * t; }
    for (i=0;i<n2;i++)
      { x[4*i+n2] = x[4*i+n2]+x[4*i-1+n1];}
    for(i=0;i<n2;i++)
      { x[i+n1] = x[i+n1]+x[i+n2+n1]; }
    x[n1+n2] = x[n2+n1]+x[n1+n2-1]-x[n-1];
    for (j=0,i=0;i<(n-1);i++)
      { if ( i < j )
          { t = x[j];
            x[j] = x[i];
            x[i] = t;
          }
        k = n/2;
        while ( k < (j+1) )
          { j = j-k;
            k = k/2;
          }
        j = j+k;
      }
    x[0] *= 1.0/sqrt((double)n);
    t = sqrt(2.0/n);
    for (i=1;i<n;i++)
      x[i] *= t;
  }
```

五、例　题

设 $n=8$，计算序列 $x(i)=e^{-0.5i}(i=0,1,\cdots,n-1)$的离散余弦变换。

主函数程序(文件名:fct8.m):

```
  #include "math.h"
  #include "fct8.c"
  main()
  { int i,n;
    double x[8];
```

```
    n=8;
    for (i=0;i<n;i++)
      x[i]=exp(-0.5*i);
    fct8(x,n);
    printf("\n Discrete Cosine Transform\n");
    for (i=0;i<n;i+=4)
      { printf("      %10.7f      %10.7f",x[i],x[i+1]);
        printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
        printf("\n");
      }
  }
```

运行结果：

序列 $x(i)$ 的离散余弦变换为

0.8820963	0.7949952	0.3497837	0.1843335
0.0998366	0.0607510	0.0332076	0.0157052

§2.7　$N=8$ 点快速离散余弦反变换

一、功　能

用 B.G.Lee 算法计算 $N=8$ 点离散余弦反变换。

二、方法简介

实序列 $X(k)$ 的离散余弦反变换为

$$x(n)=\sqrt{\frac{2}{N}}\sum_{k=0}^{N-1}c(k)X(k)\cos\left[\frac{(2n+1)k\pi}{2N}\right]\quad,\quad n=0,1,\cdots,N-1$$

其中

$$c(k)=\begin{cases}1/\sqrt{2} & ,\quad k=0\\ 1 & ,\quad k=1,2,\cdots,N-1\end{cases}$$

记

$$\overline{X}(k)=\sqrt{\frac{2}{N}}c(k)X(k)\quad,\quad k=0,1,\cdots,N-1$$

B.G.Lee 提出的离散余弦反变换的快速算法如下：

(1) 计算

$$\begin{cases}G(k)=\overline{X}(2k)\\ H(k)=\overline{X}(2k+1)+\overline{X}(2k-1)\end{cases}$$
$$k=0,1,\cdots,N/2-1$$

(2) 计算两个 $N/2$ 点的离散余弦反变换

$$\begin{cases} g(n) = \sum_{k=0}^{N/2-1} G(k)\cos\dfrac{(2n+1)k\pi}{N} \\ h(n) = \sum_{k=0}^{N/2-1} H(k)\cos\dfrac{(2n+1)k\pi}{N} \end{cases}$$

$$n = 0,1,\cdots,N/2-1$$

(3) 蝶形运算

$$\begin{cases} x(n) = g(n) + h(n)/2\cos\dfrac{(2n+1)\pi}{2N} \\ x(N-n-1) = g(n) - h(n)/2\cos\dfrac{(2n+1)\pi}{2N} \end{cases}$$

$$n = 0,1,\cdots,N/2-1$$

这样就把一个 N 点 IDCT 分解为两个 $N/2$ 点的 IDCT。依此类推，直至分解到最后一级为止。

三、使用说明

1. 子函数语句

void ifct8(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:ifct8.c)

```
#include "math.h"
void ifct8(x,n)
int n;
double x[8];
{ int i,j,k,n1,n2;
  double t,pi,coef[8];
  n1 = n/2;
  n2 = n/4;
  pi = 3.141592654;
  x[0] = x[0]/sqrt(2.0);
  for(i=0;i<n2;i++)
    { coef[i] = 1/(2*cos((2*i+1)*pi/(2*n)));
      coef[i+n1] = 1/(2*cos((2*i+1)*2*pi/(2*n)));
      coef[i+n2] = 1/(2*cos((n-(2*i+1))*pi/(2*n)));
    }
```

```
for (j=0,i=0;i<(n-1);i++)
  { if ( i < j )
      { t = x[j];
        x[j] = x[i];
        x[i] = t;
      }
    k = n/2;
    while ( k < (j+1) )
      { j = j-k;
        k = k/2;
      }
    j = j+k;
  }
t = x[n1+n2-1] + x[n1+n2];
for(i=0;i<n2;i++)
  { x[i+n1+n2] = x[i+n1]+x[i+n1+n2]; }
x[n1+n2-1] = t;
for (i=0;i<n2;i++)
  { x[4*i+n1-1] = x[4*i+n1-1]+x[4*i+n1-2];}
t = 1/(2*cos((4*pi/(2*n))));
for (i=1;i<n;i+=2)
  { x[i] = x[i]*t; }
for (i=0;i<n1;i++)
  { t = x[2*i] + x[2*i+1];
    x[2*i+1] = x[2*i] - x[2*i+1];
    x[2*i] = t;
  }
for (i=0;i<n2;i++)
for (j=0;j<n2;j++)
  { x[4*i+j+n2] = x[4*i+j+n2]*coef[j+n1]; }
for (i=0;i<n2;i++)
for (j=0;j<n2;j++)
  { t = x[4*i+j] + x[4*i+j+n2];
    x[4*i+j+n2] = x[4*i+j]    x[4*i+j+n2];
    x[4*i+j] = t;
  }
for (i=0;i<n2;i++)
  { x[i+n1] = x[i+n1]*coef[i];
```

```
        x[i+n1+n2] = x[i+n1+n2] * coef[i+n2];
      }
    for (i=0;i<n1;i++)
      { t = x[i] + x[i+n1];
        x[i+n1] = x[i] - x[i+n1];
        x[i] = t;
      }
    t = x[2]; x[2] = x[3]; x[3] = t;
    t = x[4]; x[4] = x[6]; x[6] = x[5]; x[5] = x[7]; x[7] = t;
    t = sqrt(2.0/n);
    for (i=0;i<n;i++)
      x[i] *= t;
  }
```

五、例　题

设$n=8$，计算序列$x(i)=e^{-0.5i}(i=0,1,\cdots,n-1)$的离散余弦变换和离散余弦反变换。

主函数程序(文件名：ifct8.m)：

```
#include "math.h"
#include "fct8.c"
#include "ifct8.c"
main()
{ int i,n;
  double x[8];
  n=8;
  for (i=0;i<n;i++)
    x[i]=exp(-0.5*i);
  printf("\n Input Sequence\n");
  for (i=0;i<n;i+=4)
    { printf("      %10.7f      %10.7f",x[i],x[i+1]);
      printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
      printf("\n");
    }
  fct8(x,n);
  printf("\n Discrete Cosine Transform\n");
  for (i=0;i<n;i+=4)
    { printf("      %10.7f      %10.7f",x[i],x[i+1]);
      printf("      %10.7f      %10.7f",x[i+2],x[i+3]);
```

```
        printf("\n");
      }
    ifct8(x,n);
    printf("\n Inverse Discrete Cosine Transform\n");
    for (i=0;i<n;i+=4)
      { printf("        %10.7f        %10.7f",x[i],x[i+1]);
        printf("        %10.7f        %10.7f",x[i+2],x[i+3]);
        printf("\n");
      }
  }
```

运行结果：

输入序列 $x(i)$为

1.0000000	0.6065307	0.3678794	0.2231302
0.1353353	0.0820850	0.0497871	0.0301974

离散余弦变换 $X(k)$为

0.8820963	0.7949952	0.3497837	0.1843335
0.0998366	0.0607510	0.0332076	0.0157052

离散余弦反变换 $x(i)$为

1.0000000	0.6065307	0.3678794	0.2231302
0.1353353	0.0820850	0.0497871	0.0301974

§2.8 快速离散正弦变换

一、功　能

用 N 点快速傅立叶变换计算离散正弦变换。

二、方法简介

实序列 $x(n)$的离散正弦变换为

$$X(k)=\sqrt{\frac{2}{N}}\sum_{n=0}^{N-1}x(n)\sin\frac{nk\pi}{N}\quad,\quad k=0,1,\cdots,N-1$$

实序列 $X(k)$的离散正弦反变换为

$$x(n)=\sqrt{\frac{2}{N}}\sum_{k=0}^{N-1}X(k)\sin\frac{nk\pi}{N}\quad,\quad n=0,1,\cdots,N-1$$

为了用 FFT 计算离散正弦变换，我们构造序列 $y(n)$如下：

$$\begin{cases}y(0)=0\\ y(n)=\sqrt{\dfrac{2}{N}}\left\{[x(n)+x(N-n)]\sin\dfrac{n\pi}{N}+\dfrac{1}{2}[x(n)-x(N-n)]\right\}\end{cases}$$

$$n = 1,2,\cdots,N-1$$

计算序列 $y(n)$的傅立叶变换 $Y_F(k)$

$$Y_F(k) = R(k) + j\,I(k)$$

其中

$$\begin{aligned} R(k) &= \sum_{n=0}^{N-1} y(n)\cos\frac{2\pi}{N}nk \\ &= \sqrt{\frac{2}{N}}\sum_{n=1}^{N-1}[x(n)+x(N-n)]\sin\frac{n\pi}{N}\cos\frac{2\pi}{N}nk \\ &= X(2k+1) - X(2k-1) \end{aligned}$$

及

$$\begin{aligned} I(k) &= -\sum_{n=0}^{N-1} y(n)\sin\frac{2\pi}{N}nk \\ &= -\sqrt{\frac{2}{N}}\sum_{n=1}^{N-1}[x(n)-x(N-n)]\frac{1}{2}\sin\frac{2\pi}{N}nk \\ &= -X(2k) \end{aligned}$$

因此，离散正弦变换 $X(k)$可由下式计算

$$\begin{cases} X(2k) = -I(k) & ,\quad k = 1,2,\cdots,\dfrac{N}{2}-1 \\ X(2k+1) = X(2k-1) - R(k) & ,\quad k = 1,2,\cdots,\dfrac{N}{2}-1 \\ X(1) = \sqrt{\dfrac{2}{N}}\displaystyle\sum_{n=0}^{N-1}x(n)\sin\frac{n\pi}{N} \end{cases}$$

由于离散正弦变换和离散正弦反变换结构上完全相同，因此两者可以用一个程序。

三、使用说明

1. 子函数语句

void fst(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:fst.c)

```
#include "math.h"
#include "stdlib.h"
#include "rfft.c"
void fst(x,n)
int n;
```

```
double x[];
{ int i,n1;
  double q,s,x1,*y;
  y = malloc(n*sizeof(double));
  n1 = n/2;
  q = 4.*atan(1.0)/n;
  y[0] = 0.0;
  for (i=1;i<n;i-+)
    y[i] = sin(i*q)*(x[i]+x[n-i])+0.5*(x[i]-x[n-i]);
  rfft(y,n);
  x1 = 0.0;
  for (i=1;i<n;i++)
    { s = i*q;
      s = sin(s);
      x1 += s*x[i];
    }
  x[0] = 0.0; x[1] = x1;
  for (i=1;i<n1;i++)
    { x[2*i] = -y[n-i];
      x[2*i+1] = x[2*i-1]+y[i];
    }
  s = sqrt(2.0/n);
  for (i=1;i<n;i++)
    x[i] *= s;
  free(y);
}
```

五、例　题

设 $n=16$，计算序列 $x(i)=e^{-0.5i}$ $(i=1,2,\cdots,n-1)$的离散正弦变换和离散正弦反变换。

主函数程序(文件名:fst.m):

```
#include "math.h"
#include "fst.c"
main()
{ int i,n;
  double x[16];
  n=16;
  for (i=0;i<n;i++)
```

```
    x[i]=exp(-0.5*i);
  printf("INPUT SEQUENCE\n");
  for (i=1;i<n;i++)
    { printf("          %10.7f",x[i]);
      if (i%4==0) printf("\n");
    }
  printf("\n");
  fst(x,n);
  printf("DISCRETE SINE TRANSFORM\n");
  for (i=1;i<n;i++)
    { printf("          %10.7f",x[i]);
      if (i%4==0) printf("\n");
    }
  printf("\n");
  fst(x,n);
  printf("INVERSE DISCRETE SINE TRANSFORM\n");
  for (i=1;i<n;i++)
    { printf("          %10.7f",x[i]);
      if (i%4==0) printf("\n");
    }
  printf("\n");
}
```

运行结果：

输入序列 $x(i)$为

```
0.6065307    0.3678794    0.2231302    0.1353353
0.0820850    0.0497871    0.0301974    0.0183156
0.0111090    0.0067379    0.0040868    0.0024788
0.0015034    0.0009119    0.0005531
```

离散正弦变换 $X(k)$为：

```
0.2349416    0.3319166    0.3317329    0.2971519
0.2570270    0.2191653    0.1859855    0.1567163
0.1311227    0.1081008    0.0873539    0.0681070
0.0501480    0.0329645    0.0163626
```

离散正弦反变换 $x(i)$为：

```
0.6065307    0.3678794    0.2231302    0.1353353
0.0820850    0.0497871    0.0301974    0.0183156
0.0111090    0.0067379    0.0040868    0.0024788
0.0015034    0.0009119    0.0005531
```

§ 2.9 快速沃尔什(Walsh)变换

一、功　能

计算实序列的离散沃尔什(Walsh)变换。

二、方法简介

离散沃尔什函数有三种形式，分别定义为：
沃尔什编号的离散沃尔什函数 $\mathrm{Wal}_W(k,n)$

$$\mathrm{Wal}_W(k,n)=(-1)^{\sum_{i=0}^{M-1} n_{M-1-i}\cdot g_i(k)}$$

哈达玛编号的离散沃尔什函数 $\mathrm{Wal}_H(k,n)$

$$\mathrm{Wal}_H(k,n)=(-1)^{\sum_{i=0}^{M-1} n_{M-1-i}\cdot k_{M-1-i}}$$

佩利编号的离散沃尔什函数 $\mathrm{Wal}_P(k,n)$

$$\mathrm{Wal}_P(k,n)=(-1)^{\sum_{i=0}^{M-1} n_{M-1-i}\cdot k_i}$$

$$n=0,1,\cdots,2^M-1 \quad , \quad k=0,1,\cdots,2^M-1$$

其中 n_i 是 n 的第 i 位二进位码，k_i 是 k 的第 i 位二进位码，$g_i(k)$是 k 的格雷(Gray)码的第 i 位码。

离散沃尔什变换定义为

$$X(k)=\frac{1}{N}\sum_{n=0}^{N-1} x(n)\ \mathrm{Wal}(k,n) \quad , \quad k=0,1,\cdots,N-1$$

离散沃尔什反变换定义为

$$x(n)=\sum_{k=0}^{N-1} X(k)\ \mathrm{Wal}(k,n) \quad , \quad n=0,1,\cdots,N-1$$

其中 $\mathrm{Wal}(k,n)$是上面三种编号的离散沃尔什函数之一。

设以哈达玛编号的沃尔什函数 $\mathrm{Wal}_H(k,n)$为元素构成的矩阵为 $[H_N]$，它有如下递推关系

$$[H_N]=\begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix}$$

利用该递推公式将变换矩阵进行稀疏矩阵分解，可以推导出与快速傅立叶变换类似的算法。

其它两种编号的沃尔什变换可以由哈达玛编号的沃尔什变换导出，此不赘述。

快速沃尔什变换完全不需要乘法，只需作 $N\log_2 N$ 次加减运算。另外，从沃尔什变换的定义可以看出，正、反变换除相差一个常数因子 $1/N$ 外，其结构完全相同，因此可用相同程序计算离散沃尔什正、反变换。

三、使用说明

1. 子函数语句

void fwt(x,n,sign)

2. 形参说明

x ——双精度实型一维数组，长度为n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是2的整数次幂，即 $n=2^m$。

sign ——整型变量。变换类型的控制参数。

当 sign=1 时，计算哈达玛排序的快速沃尔什变换；

当 sign=2 时，计算沃尔什排序的快速沃尔什变换；

当 sign=3 时，计算佩利排序的快速沃尔什变换。

四、子函数程序(文件名:fwt.c)

```
void fwt(x,n,sign)
int n,sign;
double x[];
{ int i,j,j1,k,k1,l,m,n0,n1,n2,flag;
  double t;
  for (j=1,i=1;i<16;i++)
    { m = i;
      j = 2*j;
      if (j==n) break;
    }
  if ( sign >= 2 )
    { n1 = n-1;
      for (j=0,i=0;i<n1;i++)
        { if ( i < j )
            { t = x[j];
              x[j] = x[i];
              x[i] = t;
            }
          k = n/2;
          while ( k < (j+1) )
            { j = j-k;
              k = k/2;
            }
          j = j+k;
```

```
        }
      }
    n2 = 1;
    for (l=1;l<=m;l++)
      { n0 = n / n2;
        n1 = n0 / 2;
        flag = 1;
        for (j=0;j<n2;j++)
          { j1 = j * n0;
            for (i=0;i<n1;i++)
              { k = i + j1;
                k1 = k + n1;
                t = x[k1] * flag;
                x[k1] = x[k] - t;
                x[k] = x[k] + t;
              }
            if ( sign == 2 ) flag = - flag;
          }
        n2 = n2 * 2;
      }
    for (i=0;i<n;i++)
      { x[i] /= n; }
}
```

五、例　题

计算 $n=8$ 点序列{ 1,2,1,1,3,2,1,2 }的哈达玛排序的、沃尔什排序的以及佩利排序的离散沃尔什变换，并计算它们的反变换。

主函数程序(文件名:fwt.m)：

```
#include "fwt.c"
main()
{ int i,n;
  static double x[8]={1.,2.,1.,1.,3.,2.,1.,2.};
  n=8;
  fwt(x,n,1);
  printf("\n Hadamard-ordered Walsh Transform\n");
  for (i=0;i<n;i+=4)
    { printf("     %10.7f      %10.7f",x[i],x[i+1]);
      printf("     %10.7f      %10.7f",x[i+2],x[i+3]);
```

```
    printf("\n");
  }
fwt(x,n,1);
for (i=0;i<n;i++)
  x[i] = n * x[i];
printf("\n Inverse Hadamard-ordered Walsh Transform\n");
for (i=0;i<n;i+=4)
  { printf("      %10.7f        %10.7f",x[i],x[i+1]);
    printf("      %10.7f        %10.7f",x[i+2],x[i+3]);
    printf("\n");
  }
fwt(x,n,2);
printf("\n Walsh-ordered Walsh Transform\n");
for (i=0;i<n;i+=4)
  { printf("      %10.7f        %10.7f",x[i],x[i+1]);
    printf("      %10.7f        %10.7f",x[i+2],x[i+3]);
    printf("\n");
  }
fwt(x,n,2);
for (i=0;i<n;i++)
  x[i] = n * x[i];
printf("\n Inverse Walsh-ordered Walsh Transform\n");
for (i=0;i<n;i+=4)
  { printf("      %10.7f        %10.7f",x[i],x[i+1]);
    printf("      %10.7f        %10.7f",x[i+2],x[i+3]);
    printf("\n");
  }
fwt(x,n,3);
printf("\n Paley-ordered Walsh Transform\n");
for (i=0;i<n;i+=4)
  { printf("      %10.7f        %10.7f",x[i],x[i+1]);
    printf("      %10.7f        %10.7f",x[i+2],x[i+3]);
    printf("\n");
  }
fwt(x,n,3);
for (i=0;i<n;i++)
  x[i] = n * x[i];
printf("\n Inverse Paley-ordered Walsh Transform\n");
```

```
    for (i=0;i<n;i+=4)
      { printf("        %10.7f        %10.7f",x[i],x[i+1]);
        printf("        %10.7f        %10.7f",x[i+2],x[i+3]);
        printf("\n");
      }
  }
```

运行结果：

哈达玛排序的沃尔什变换为

```
      1.6250000      -0.1250000       0.3750000       0.1250000
     -0.3750000      -0.1250000      -0.1250000      -0.3750000
```

哈达玛排序的沃尔什反变换为

```
      1.0000000       2.0000000       1.0000000       1.0000000
      3.0000000       2.0000000       1.0000000       2.0000000
```

沃尔什排序的沃尔什变换为

```
      1.6250000      -0.3750000      -0.1250000       0.3750000
      0.1250000      -0.3750000      -0.1250000      -0.1250000
```

沃尔什排序的沃尔什反变换为

```
      1.0000000       2.0000000       1.0000000       1.0000000
      3.0000000       2.0000000       1.0000000       2.0000000
```

佩利排序的沃尔什变换为

```
      1.6250000      -0.3750000       0.3750000      -0.1250000
     -0.1250000      -0.1250000       0.1250000      -0.3750000
```

佩利排序的沃尔什反变换为

```
      1.0000000       2.0000000       1.0000000       1.0000000
      3.0000000       2.0000000       1.0000000       2.0000000
```

§2.10　快速希尔伯特变换(一)

一、功　能

用快速哈特莱(Hartley)变换计算离散希尔伯特(Hilbert)变换。

二、方法简介

设离散希尔伯特变换的变换核为 $h(n)$，那么序列 $x(n)$ 的离散希尔伯特变换 $\hat{x}(n)$ 定义为 $x(n)$ 与 $h(n)$ 的卷积，即

$$\hat{x}(n) = x(n) * h(n)$$

设序列 $\hat{x}(n)$、$x(n)$ 和 $h(n)$ 的离散哈特莱变换分别为 $\hat{X}(k)$、$X(k)$ 和 $H(k)$，由上式可

得

$$\hat{X}(k) = X(k)H_e(k) + X(-k)H_o(k)$$

其中 $H_e(k)$和 $H_o(k)$分别是 $H(k)$的偶部和奇部。

由于希尔伯特变换核 $h(n)$是奇函数，从而有

$$\begin{cases} H_e(k) = 0 \\ H_o(k) - H(k) \end{cases}$$

这样 $\hat{X}(k)$可表示为

$$\hat{X}(k) = X(-k)H(K)$$

其中

$$H(k) = \begin{cases} 1 \quad , & k = 1,2,\cdots,\frac{N}{2}-1 \\ 0 \quad , & k = 0,\frac{N}{2} \\ -1 \quad , & k = \frac{N}{2}+1,\frac{N}{2}+2,\cdots,N-1 \end{cases}$$

而 $X(-k)$可表示为

$$X(-k) = \begin{cases} X(k) \quad , & k = 0 \\ X(N-k) \quad , & k = 1,2,\cdots,N-1 \end{cases}$$

最后，计算 $\hat{X}(k)$的离散哈特莱反变换(IDHT)，即可得到序列 $x(n)$的离散希尔伯特变换 $\hat{x}(n)$。

三、使用说明

1. 子函数语句

void hilbth(x,n)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:hilbth.c)

```
#include "fht.c"
void hilbth(x,n)
int n;
double x[];
{ int i,n1,n2;
  double t;
  n1 = n/2;
  n2 = n1+1;
  fht(x,n);
```

```
for (i=1;i<n1;i++)
  { t = x[i];
    x[i] = x[n-i];
    x[n-i] = t;
  }
for (i=n2;i<n;i++)
  x[i] = - x[i];
x[0] = 0.0;
x[n1] = 0.0;
fht(x,n);
t = 1.0/n;
for (i=0;i<n;i++)
  x[i] *= t;
}
```

五、例　题

设 $n=64$，序列 $x(i)=\sin(2\pi i/32)$，$i=0,1,\cdots,n-1$。计算其离散希尔伯特变换 $\hat{x}(i)$，并画出 $x(i)$ 与 $\hat{x}(i)$ 的图形。

主函数程序(文件名：hilbth.m)：

```
#include "math.h"
#include "stdio.h"
#include "hilbth.c"
main()
{ int i,n;
  double x[64];
  FILE *fp;
  n = 64;
  for (i=0;i<n;i++)
    x[i] = sin(2*3.14159265*i/32);
  fp = fopen("hilb1.dat","w");
  for (i=0;i<n;i++)
    fprintf(fp,"%d %f\n",i,x[i]);
  fclose(fp);
  printf("\n Original Sequence\n");
  for (i=0;i<n/2;i+=4)
    { printf("       %10.7f       %10.7f",x[i],x[i+1]);
      printf("       %10.7f       %10.7f\n",x[i+2],x[i+3]);
    }
```

```
  hilbth(x,n);
  printf("\n Discrete Hilbert Transform\n");
  for (i=0;i<n/2;i-=4)
    { printf("        %10.7f        %10.7f",x[i],x[i+1]);
      printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
    }
  fp = fopen("hilb2.dat","w");
  for (i=0;i<n;i++)
    fprintf(fp,"%d %f\n",i,x[i]);
  fclose(fp);
}
```

运行结果：

输入序列 $x(i)$（前 32 个数据）

0.0000000	0.1950903	0.3826834	0.5555702
0.7071068	0.8314696	0.9238795	0.9807853
1.0000000	0.9807853	0.9238795	0.8314696
0.7071068	0.5555702	0.3826834	0.1950903
0.0000000	−0.1950903	−0.3826834	−0.5555702
−0.7071068	−0.8314696	−0.9238795	−0.9807853
−1.0000000	−0.9807853	−0.9238795	−0.8314696
−0.7071068	−0.5555702	−0.3826834	−0.1950903

序列 $x(i)$的离散希尔伯特变换 $\hat{x}(i)$（前 32 个数据）

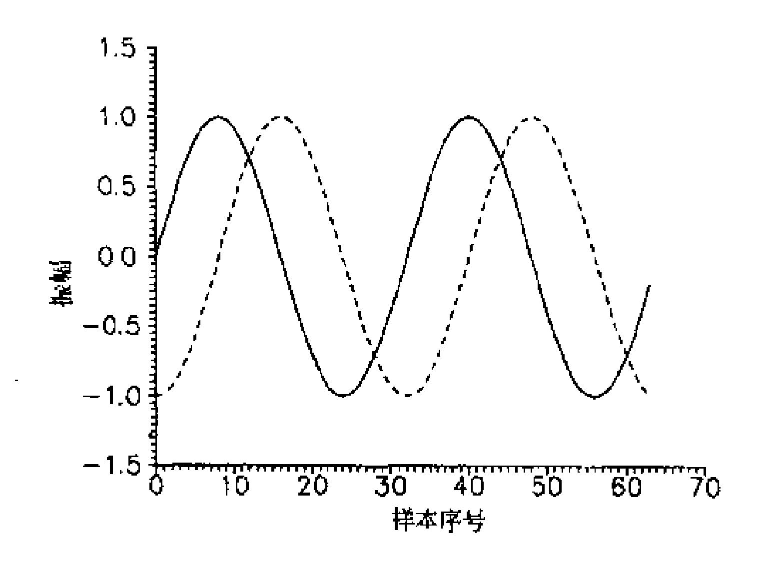
−1.0000000	−0.9807853	−0.9238795	−0.8314696
−0.7071068	−0.5555702	−0.3826834	−0.1950903
−0.0000000	0.1950903	0.3826834	0.5555702
0.7071068	0.8314696	0.9238795	0.9807853
1.000000	0.9807853	0.9238795	0.8314696
0.7071068	0.5555702	0.3826834	0.1950903
0.0000000	−0.1950903	−0.3826834	−0.5555702
−0.7071068	−0.8314696	−0.9238795	−0.9807853

序列 $x(i)$与其离散希尔伯特变换 $\hat{x}(i)$如图 2—2—1 所示。显然，两者的相位相差 90 度。

§2.11　快速希尔伯特变换（二）

一、功　能

用快速傅立叶变换计算离散希尔伯特(Hilbert)变换。



实线：$x(i)$　虚线：希尔伯特变换 $\hat{x}(i)$

图 2－2－1　序列 $x(i)$和希尔伯特变换 $\hat{x}(i)$

二、方法简介

设离散希尔伯特变换的变换核为 $h(n)$，那么序列 $x(n)$ 的离散希尔伯特变换 $\hat{x}(n)$ 定义为 $x(n)$与 $h(n)$的卷积，即

$$\hat{x}(n) = x(n) * h(n)$$

设序列 $\hat{x}(n)$、$x(n)$和 $h(n)$的离散傅立叶变换分别为 $\hat{X}(k)$、$X(k)$和 $H(k)$，由上式可得

$$\hat{X}(k) = X(k)H(k)$$

对上式进行傅立叶反变换，便可得到离散希尔伯特变换 $\hat{x}(n)$

$$\hat{x}(n) = \mathrm{IDFT}[X(k)H(k)]$$

其中

$$H(k) = \begin{cases} -j \ , & k = 1,2,\cdots,\frac{N}{2}-1 \\ 0 \ , & k = 0,\frac{N}{2} \\ j \ , & k = \frac{N}{2}+1,\frac{N}{2}+2,\cdots,N-1 \end{cases}$$

三、使用说明

1. 子函数语句

```
void hilbtf(x,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放要变换的实数据，最后存放变换结果。

n ——整型变量。数据长度，必须是 2 的整数次幂，即 $n=2^m$。

四、子函数程序(文件名:hilbtf.c)

```
#include "rfft.c"
#include "irfft.c"
void hilbtf(x,n)
int n;
double x[];
{ int i,n1;
  double t;
  n1 = n/2;
  rfft(x,n);
  for(i=1;i<n1;i++)
    { t = x[i];
      x[i] = x[n-i];
      x[n-i] = -t;
    }
   x[0]=x[n1]=0.0;
   irfft(x,n);
 }
```

五、例 题

设 $n=64$，序列 $x(i)=\sin(2\pi i/n)$，$i=0,1,\cdots,n-1$，它的理想希尔伯特变换为 $\hat{x}(i)=-\cos(2\pi i/n)$，$i=0,1,\cdots,n-1$。用快速傅立叶变换计算序列 $x(i)$的离散希尔伯特变换 $\hat{x}(i)$，并与理想值进行比较。

主函数程序(文件名:hilbtf.m)：

```
 #include "math.h"
 #include "hilbtf.c"
 main()
 { int i,n;
   double x[64],y[64];
   n = 64;
   for (i=0;i<n;i++)
     x[i] = sin(2*3.14159265*i/n);
   for (i=0;i<n;i+-)
     y[i] = -cos(2*3.14159265*i/n);
   printf("\n Ideal Discrete Hilbert Transform\n");
   for (i=0;i<n/2;i+=4)
```

```
    { printf("        %10.7f        %10.7f",y[i],y[i+1]);
      printf("        %10.7f        %10.7f\n",y[i+2],y[i+3]);
    }
  hilbtf(x,n);
  printf("\n Discrete Hilbert Transform\n");
  for (i=0;i<n/2;i+=4)
    { printf("        %10.7f        %10.7f",x[i],x[i+1]);
      printf("        %10.7f        %10.7f\n",x[i+2],x[i+3]);
    }
}
```

运行结果：

序列 $x(i)$的理想离散希尔伯特变换(前 32 个数据)

−1.0000000	−0.9951847	−0.9807853	−0.9569403
−0.9238795	−0.8819213	−0.8314696	−0.7730105
−0.7071068	−0.6343933	−0.5555702	−0.4713967
−0.3826834	−0.2902847	−0.1950903	−0.0980171
−0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847

序列 $x(i)$的离散希尔伯特变换(前 32 个数据)

−1.0000000	−0.9951847	−0.9807853	−0.9569403
−0.9238795	−0.8819213	−0.8314696	−0.7730105
−0.7071068	−0.6343933	−0.5555702	−0.4713967
−0.3826834	−0.2902847	−0.1950903	−0.0980171
−0.0000000	0.0980171	0.1950903	0.2902847
0.3826834	0.4713967	0.5555702	0.6343933
0.7071068	0.7730105	0.8314696	0.8819213
0.9238795	0.9569403	0.9807853	0.9951847

第三章　快速卷积与相关

§3.1　快速卷积

一、功　能

用快速傅立叶变换计算两个有限长序列的线性卷积。

二、方法简介

设序列 $x(n)$的长度为 M，序列 $y(n)$的长度为 N，序列 $x(n)$与 $y(n)$的线性卷积定义为

$$z(n)=\sum_{i=0}^{M-1}x(i)y(n-i)\ ,\quad n=0,1,\cdots,M-N-2$$

用快速傅立叶变换计算线性卷积的算法如下

1. 选择 L 满足下述条件

$$\begin{cases}L\geqslant M+N-1\\ L=2^{\gamma},\quad \gamma\text{ 为正整数}\end{cases}$$

2. 将序列 $x(n)$与 $y(n)$按如下方式补零，形成长为 $L=2^{\gamma}$ 的序列

$$x(n)=\begin{cases}x(n) & ,\quad n=0,1,\cdots,M-1\\ 0 & ,\quad n=M,M+1,\cdots,L-1\end{cases}$$

$$y(n)=\begin{cases}y(n) & ,\quad n=0,1,\cdots,N-1\\ 0 & ,\quad n=N,N+1,\cdots,L-1\end{cases}$$

3. 用 FFT 算法分别计算 $x(n)$与 $y(n)$的离散傅立叶变换 $X(k)$与 $Y(k)$

$$X(k)=\sum_{n=0}^{L-1}x(n)e^{-j2\pi nk/L}$$

$$Y(k)=\sum_{n=0}^{L-1}y(n)e^{-j2\pi nk/L}$$

4. 计算 $X(k)$与 $Y(k)$的乘积

$$Z(k)=X(k)Y(k)$$

5. 用 FFT 算法计算 $Z(k)$的离散傅立叶反变换，得到卷积 $z(n)$

$$z(n)=\frac{1}{L}\sum_{k=0}^{L-1}Z(k)e^{j2\pi nk/L},\quad n=0,1,\cdots,L-1$$

序列 $z(n)$的前 $M+N-1$ 点的值就是序列 $x(n)$与 $y(n)$的线性卷积。

三、使用说明

1. 子函数语句

void convol(x,y,m,n,len)

2. 形参说明

x ——双精度实型一维数组，长度为 len。开始时存放实序列 $x(i)$，最后存放线性卷积的值。

y —— 双精度实型一维数组，长度为 n。存放实序列 $y(i)$。

m —— 整型变量。序列 $x(i)$的长度。

n —— 整型变量。序列 $y(i)$的长度。

len —— 整型变量，线性卷积的长度。len≥m+n−1，且必须是 2 的整数次幂，即 $len=2^{\gamma}$。

四、子函数程序(文件名:convol.c)

```
#include "rfft.c"
#include "irfft.c"
void convol(x,y,m,n,len)
int m,n,len;
double x[],y[];
{ int i,len2;
  double  t;
  for (i=m;i<len;i++)
    { x[i] = 0.0; }
  for (i=n;i<len;i++)
    { y[i] = 0.0; }
  rfft(x,len);
  rfft(y,len);
  len2 = len/2;
  x[0]      = x[0]*y[0];
  x[len2] = x[len2]*y[len2];
  for (i=1;i<len2;i++)
    { t          = x[i]*y[i]      - x[len-i]*y[len-i];
      x[len-i] = x[i]*y[len-i] + x[len-i]*y[i];
      x[i]      = t;
    }
  irfft(x,len);
}
```

五、例　题

设序列 $x(i)$ 为{−1,5,2,7,6,9,4}，其长度 $m=7$；序列 $y(i)$ 为{1,2,1,−3,4,5}，其长度 $n=6$。选择 len=16，用 FFT 计算序列 $x(i)$ 与 $y(i)$ 的线性卷积，并与直接计算结果进行比较。

主函数程序（文件名：convol.m）：

```
#include "convol.c"
main()
{ int i,j,m,n,len,len1;
  double z[16];
  double x[16]={-1,5,2,7,6,9,4}, y[16]={1,2,1,-3,4,5};
  m = 7;
  n = 6;
  len = 16;
  len1 = m + n - 1;
  for (j=0;j<len1;j++)
    { z[j] = 0.0;
      for (i=0;i<m;i++)
        { if ( ( (j-i) >= 0 ) && ( (j-i) < n )  )
              z[j] += x[i] * y[j-i];
        }
    }
  printf("Direct Calculation of Linear Convolution\n");
  for (i=0;i<len1;i++)
    { printf("        %10.1lf",z[i]);
      if (i%4==3) printf("\n");
    }
  printf("\n");
  convol(x,y,m,n,len);
  printf("Fast Calculation of Linear Convolution\n");
  for (i=0;i<len1;i++)
    { printf("        %10.1lf",x[i])
      if (i%4==3) printf("\n");
    }
  printf("\n");
}
```

运行结果：

线性卷积的直接计算结果

−1.0	3.0	11.0	19.0
3.0	37.0	40.0	37.0
36.0	54.0	61.0	20.0

线性卷积的快速算法结果

−1	3.0	11.0	19.0
3.0	37.0	40.0	37.0
36.0	54.0	61.0	20.0

§3.2 长序列的快速卷积

一、功　能

用重叠保留法和快速傅立叶变换计算一个长序列和一个短序列的线性卷积。它通常用于数字滤波。

二、方法简介

设长序列 $x(n)$ 的长度为 L，序列 $h(n)$ 的长度为 M，序列 $x(n)$ 与 $h(n)$ 的卷积定义为

$$y(n) = \sum_{i=0}^{L-1} x(i)h(n-i)$$

用重叠保留法和快速傅立叶变换计算线性卷积的算法如下：

1. 将序列 $h(n)$ 按如下方式补零，形成长度为 $N=2^{\gamma}$ 的序列

$$h(n) = \begin{cases} h(n) & , \quad n = 0,1,\cdots,M-1 \\ 0 & , \quad n = M,M+1,\cdots,N-1 \end{cases}$$

2. 用 FFT 算法计算序列 $h(n)$ 的离散傅立叶变换 $H(k)$

$$H(k) = \sum_{n=0}^{N-1} h(n)e^{-j2\pi nk/N}$$

3. 将长序列 $x(n)$ 分为长度为 N 的小段 $x_i(n)$，相邻段间重叠 $M-1$ 点，即

$$x_i(n) = \begin{cases} x[n-i(N-M+1)-(M-1)] & , \quad 0 \leqslant n \leqslant N-1 \\ 0 & , \quad n \text{ 为其他值} \end{cases}$$

$$i = 0,1,\cdots$$

注意：对于第一段的 $x_0(n)$，由于没有前一段的保留信号，因此要在其前面填补 $M-1$ 个零。

4. 用 FFT 算法计算序列 $x_i(n)$ 的离散傅立叶变换 $X_i(k)$

$$X_i(k) = \sum_{n=0}^{N-1} x_i(n)e^{-j2\pi nk/N}$$

5. 计算 $X_i(k)$ 与 $H(k)$ 的乘积

$$Y_i(k) = X_i(k)H(k)$$

6. 用 FFT 算法计算 $Y_i(k)$ 的离散傅立叶反变换，得到序列 $x_i(n)$ 与 $h(n)$ 的卷积 $y_i(n)$

$$y_i(n) = \frac{1}{N}\sum_{k=0}^{N-1} Y_i(k) e^{j2\pi nk/N}, \quad n = 0,1,\cdots,N-1$$

7. 将 $y_i(n)$的前 $M-1$ 点舍去，仅保留后面的 $N-M+1$ 个样本。

$$\bar{y}_i(n) = \begin{cases} y_i(n) & , \quad M-1 \leqslant n \leqslant N-1 \\ 0 & , \quad n \text{ 为其他值} \end{cases}$$

8. 重复步骤 3～7，直到所有分段算完为止。

9. 考虑到 $x(n)$分段时，$x_i(n)$有 $M-1$ 点与前一段重叠，新添的样本只有 $N-M+1$ 个，所以 $y(n)$由 $\bar{y}_i(n)$首尾衔接而成，即

$$y(n) = \sum_{i=0}^{\infty} \bar{y}_i[n - i(N-M+1) + (M-1)]$$

三、使用说明

1. 子函数语句

void convols(x,h,len,m,n)

2. 形参说明

x ——双精度实型一维数组，长度为 len。开始时存放长序列 $x(i)$，最后存放线性卷积的值。

h —— 双精度实型一维数组，长度为 m。存放短序列 $h(i)$。

len —— 整型变量。长序列 $x(i)$的长度。

m —— 整型变量。短序列 $h(i)$的长度。

n —— 整型变量。对长序列 $x(i)$进行分段的长度。一般选取 n 大于短序列 $h(i)$长度 m 的两倍以上，且必须是 2 的整数次幂，即 $n=2^\gamma$。

四、子函数程序(文件名:convols.c)

```
#include "math.h"
#include "stdlib.h"
#include "rfft.c"
#include "irfft.c"
void convols(x,h,len,m,n)
int m,n,len;
double h[],x[];
{ int i,j,i1,n2,num,nblks;
  double t, *r, *s;
  r = malloc(n * sizeof(double));
  s = malloc((n-m+1) * sizeof(double));
  n2 = n/2;
  num = n - m + 1;
  nblks = floor((len-n+m)/(double)num) + 1;
```

```
for (i=m;i<n;i++)
  { h[i] = 0.0; }
rfft(h,n);
for (j=0;j<nblks;j++)
  { if (j == 0 )
      { for (i=0;i<(m-1);i++)
          { r[i] = 0.0; }
        for (i=(m-1);i<n;i++)
          { i1 = i - m + 1;
            r[i] = x[i1];
          }
      }
    else
      { for (i=0;i<n;i++)
          { i1 = i + j*num - m + 1;
            r[i] = x[i1];
          }
        for (i=0;i<num;i++)
          { i1 = i + (j-1)*num;
            x[i1] = s[i];
          }
      }
    rfft(r,n);
    r[0] = r[0]*h[0];
    r[n2] = r[n2]*h[n2];
    for (i=1;i<n2;i++)
      { t = h[i]*r[i] - h[n-i]*r[n-i];
        r[n-i] = h[i]*r[n-i] + h[n-i]*r[i];
        r[i] = t;
      }
    irfft(r,n);
    for (i=(m-1);i<n;i++)
      { i1 = i - m + 1;
        s[i1] = r[i];
      }
  }
 for (i=0;i<num;i++)
   { i1 = i + (j-1)*num;
```

```
        x[i1] = s[i];
      }
    i1 = j * num;
    for (i=i1;i<len;i++)
      { x[i] = 0.0; }
    free(r);
    free(s);
}
```

五、例　题

设序列 $h(i) = -\delta(i-1), i = 0,1,\cdots,18$；序列 $x(i) = e^{-0.01i}, i = 0,1,\cdots,1023$；分段长度 n=64。计算序列 $h(i)$与 $x(i)$的线性卷积 $y(i)$，并将序列 $x(i)$、$y(i)$及误差 $e(i)$分别存盘，同时把序列 $x(i)$和 $y(i)$的前 32 个值显示出来。

显而易见，卷积 $y(i)$为：$y(i) = -x(i-1)$。

主函数程序(文件名:convols. m)：

```
#include "math.h"
#include "stdio.h"
#include "convols.c"
main()
{ int i,m,n,len;
  double h[64],x[1024],y[1024],e[1024];
  FILE *fp;
  n=64;
  m=19;
  len=1024;
  for (i=0;i<m;i++)
    { h[i]=0.0; }
  h[1] = -1;
  for (i=0;i<len;i++)
    { x[i]= exp(-0.01*i); }
  for (i=0;i<(len-1);i++)
    { y[i+1]=x[i]; }
  printf("Input Sequence\n");
  for (i=0;i<32;i++)
    { printf("      %10.7f",x[i]);
      if (i%4==3) printf("\n");
    }
  fp=fopen("convx.dat","w");
```

```
    for (i=0;i<len;i++)
      { fprintf(fp,"      %10.7f",x[i]);
        if (i%4==3) fprintf(fp,"\n");
      }
    fprintf(fp,"\n");
    fclose(fp);
    convols(x,h,len,m,n);
    printf("Linear Convolution\n");
    for (i=0;i<32;i++)
      { printf("        %10.7f",x[i]);
        if (i%4==3) printf("\n");
      }
    fp=fopen("convy.dat","w");
    fprintf(fp,"\n");
    for (i=0;i<len;i++)
      { fprintf(fp,"      %10.7f",x[i]);
        if (i%4==3) fprintf(fp,"\n");
      }
    fprintf(fp,"\n");
    fclose(fp);
    for (i=1;i<len;i++)
      { e[i]=fabs(y[i]+x[i]); }
    fp=fopen("conve.dat","w");
    fprintf(fp,"                 ");
    for (i=1;i<len;i++)
      { fprintf(fp,"      %10.7f",e[i]);
        if (i%4==3) fprintf(fp,"\n");
      }
    fprintf(fp,"\n");
    fclose(fp);
  }
```

运行结果：

输入序列 $x(i)$ 为

1.0000000	0.9900498	0.9801987	0.9704455
0.9607894	0.9512294	0.9417645	0.9323938
0.9231163	0.9139312	0.9048374	0.8958341
0.8869204	0.8780954	0.8693582	0.8607080
0.8521438	0.8436648	0.8352702	0.8269591

0.8187308	0.8105842	0.8025188	0.7945336
0.7866279	0.7788008	0.7710516	0.7633795
0.7557837	0.7482636	0.7408182	0.7334470

线性卷积 $y(i)$ 为

0.0000000	−1.000000	−0.9900498	−0.9801987
−0.9704455	−0.9607894	−0.9512294	−0.9417645
−0.9323938	−0.9231163	−0.9139312	−0.9048374
−0.8958341	−0.8869204	−0.8780954	−0.8693582
−0.8607080	−0.8521438	−0.8436648	−0.8352702
−0.8269591	−0.8187308	−0.8105842	−0.8025188
−0.7945336	−0.7866279	−0.7788008	−0.7710516
−0.7633795	−0.7557837	−0.7482636	−0.7408182

§3.3 特别长序列的快速卷积

一、功　能

用重叠保留法和快速傅立叶变换计算一个特别长序列和一个短序列的线性卷积。它通常用于数字滤波。

二、方法简介

设特别长序列 $x(n)$ 的长度为 L，序列 $h(n)$ 的长度为 M，序列 $x(n)$ 与 $h(n)$ 的卷积定义为

$$y(n)=\sum_{i=0}^{L-1}x(i)h(n-i)$$

用重叠保留法和快速傅立叶变换计算线性卷积的算法如下：

1. 将序列 $h(n)$ 按如下方式补零，形成长度为 $N=2^\gamma$ 的序列

$$h(n)=\begin{cases}h(n) & , \quad n=0,1,\cdots,M-1\\ 0 & , \quad n=M,M+1,\cdots,N-1\end{cases}$$

2. 用 FFT 算法计算序列 $h(n)$ 的离散傅立叶变换 $H(k)$

$$H(k)=\sum_{n=0}^{N-1}h(n)e^{-j2\pi nk/N}$$

3. 将特别长序列 $x(n)$ 分为长度为 N 的小段 $x_i(n)$，相邻段间重叠 $M-1$ 点，即

$$x_i(n)=\begin{cases}x[n+i(N-M+1)-(M-1)] & , \quad 0\leqslant n\leqslant N-1\\ 0 & , \quad n\text{ 为其他值}\end{cases}$$

$$i=0,1,\cdots$$

注意：对于第一段的 $x(n)$，由于没有前一段的保留信号，因此要在其前面填补 $M-1$ 个零。

4. 用 FFT 算法计算序列 $x_i(n)$ 的离散傅立叶变换 $X_i(k)$

$$X_i(k) = \sum_{n=0}^{N-1} x_i(n) e^{-j2\pi nk/N}$$

5. 计算 $X_i(k)$ 与 $H(k)$ 的乘积

$$Y_i(k) = X_i(k)H(k)$$

6. 用 FFT 算法计算 $Y_i(k)$ 的离散傅立叶反变换，得到序列 $x_i(n)$ 与 $h(n)$ 的卷积 $y_i(n)$

$$y_i(n) = \frac{1}{N}\sum_{k=0}^{N-1} Y_i(k) e^{j2\pi nk/N}, \quad n = 0,1,\cdots,N-1$$

7. 将 $y_i(n)$ 的前 $M-1$ 点舍去，仅保留后面的 $N-M+1$ 个样本

$$\bar{y}_i(n) = \begin{cases} y_i(n) & , \quad M-1 \leqslant n \leqslant N-1 \\ 0 & , \quad n \text{ 为其他值} \end{cases}$$

8. 重复步骤 3～7，直到所有分段算完为止。

9. 考虑到 $x(n)$ 分段时，$x_i(n)$ 有 $M-1$ 点与前一段重叠，新添的样本只有 $N-M+1$ 个，所以 $y(n)$ 由 $\bar{y}_i(n)$ 首尾衔接而成，即

$$y(n) = \sum_{i=0}^{\infty} \bar{y}_i[n - i(N-M+1) + (M-1)]$$

由于数据量很大，因此序列 $h(n)$ 和特别长序列 $x(n)$ 要存储到磁盘中，计算完毕后，卷积序列 $y(n)$ 也由程序自动存储到磁盘中。

三、使用说明

1. 子函数语句

void convold(hfname,xfname,yfname,n)

2. 形参说明

hfname ——字符型指针。它指向短序列 $h(i)$ 的文件名。

xfname ——字符型指针。它指向长序列 $x(i)$ 的文件名。

yfname ——字符型指针。它指向卷积 $y(i)$ 的文件名。

n —— 整型变量。对长序列 $x(i)$ 进行分段的长度。一般选取 n 大于短序列 $h(i)$ 长度的两倍以上，且必须是 2 的整数次幂，即 $n=2^{\gamma}$。

注意：序列 $h(i)$，$x(i)$ 和 $y(i)$ 的长度由程序自动判别。

四、子函数程序(文件名:convold.c)

```
#include "math.h"
#include "stdio.h"
#include "stdlib.h"
#include "rfft.c"
#include "irfft.c"
void convold(hfname,xfname,yfname,n)
int n;
```



```
char *hfname, *xfname, *yfname;
{ int i,j,m,i1,i2,n2,ls0,len,num,nblks;
  double t,x[5000];
  double *h, *r, *s;
  FILE *fp, *fp1, *fp2;
  r = malloc(n * sizeof(double));
  h = malloc(n * sizeof(double));
  n2 = n/2;
  fp = fopen(hfname,"r");
  i = 0;
  while (!feof(fp))
    { fscanf(fp," %lf",&h[i++]); }
  fclose(fp);
  m = i - 1;
  for (i=m;i<n;i++)
    { h[i] = 0.0; }
  rfft(h,n);
  s = malloc((n-m+1) * sizeof(double));
  num = n - m + 1;
  fp1 = fopen(xfname,"r");
  fp2 = fopen(yfname,"w");
  len = 5000;
  ls0 = 0;
  do
    { for (i=ls0;i<5000;i++)
        { if (feof(fp1))
            { len = i;
              break;
            }
          fscanf(fp1," %lf",&x[i]);
        }
      nblks = floor((len-n+m)/(double)num) + 1;
      for (j=0;j<nblks;j++)
        { if (j == 0 )
            { for (i=0;i<(m-1);i++)
                { r[i] = 0.0; }
              for (i=(m-1);i<n;i++)
                { i1 = i - m + 1;
```

```
            r[i] = x[i1];
          }
      }
    else
      { for (i=0;i<n;i++)
          { i1 = i + j*num - m + 1;
            r[i] = x[i1];
          }
        for (i=0;i<num;i++)
          { i1 = i + (j-1)*num;
            x[i1] = s[i];
          }
      }
    rfft(r,n);
    r[0] = r[0]*h[0];
    r[n2] = r[n2]*h[n2];
    for (i=1;i<n2;i++)
      { t = h[i]*r[i] - h[n-i]*r[n-i];
        r[n-i] = h[i]*r[n-i] + h[n-i]*r[i];
        r[i] = t;
      }
    irfft(r,n);
    for (i=(m-1);i<n;i++)
      { i1 = i - m + 1;
        s[i1] = r[i];
      }
  }
for (i=0;i<num;i++)
  { i1 = i + (j-1)*num;
    x[i1] = s[i];
  }
i1 = j*num;
for (i=0;i<i1;i++)
  { fprintf(fp2,"%lf\n",x[i]); }
for (i=i1;i<len;i++)
  { i2 = i-i1;
    x[i2] =x[i];
  }
```

```
      ls0 = len - i1;
    } while (! feof(fp1));
  fclose(fp1);
  fclose(fp2);
  free(r);
  free(s);
}
```

五、例　题

设序列 $h(i) = -\delta(i-1), i = 0,1,\cdots,10$；序列 $x(i) = e^{-0.01i}, i = 0,1,\cdots,9999$；分段长度 $n=64$。计算序列 $h(i)$与 $x(i)$的线性卷积 $y(i)$，并将结果存于文件 y. dat 中。

显而易见，卷积 $y(i)$为：$y(i) = -x(i-1)$。

首先，用如下程序（文件名：convin. c）在磁盘上建立数据文件 h. dat 和 x. dat。

```
#include "math.h"
#include "stdio.h"
main()
{ int i,j,len;
  double h[11],x[1000];
  FILE *fp;
  for (i=0;i<11;i++)
    { h[i]=0.0; }
  h[1]=-1.0;
  fp=fopen("h1.dat","w");
  for (i=0;i<11;i++)
    { fprintf(fp,"%lf\n",h[i]); }
  fclose(fp);
  len=1000;
  for (i=0;i<len;i++)
    { x[i]= exp(-0.01*i); }
  fp=fopen("x1.dat","w");
  for (j=0;j<10;j++)
  for (i=0;i<len;i+=1)
    { fprintf(fp,"%lf\n",x[i]); }
  fclose(fp);
}
```

主函数程序（文件名：convold. m）如下。它读取数据文件 h. dat 和 x. dat，并进行卷积计算，结果存于数据文件 y. dat 中。

```
#include "convold.c"
main()
{ int n;
  char h[]="h.dat", x[]="x.dat", y[]="y.dat";
  n=64;
  convold(h,x,y,n);
}
```

用如下程序(文件名:convout.c)读取数据文件 x.dat 和 y.dat,并将它们显示出来。

```
#include "stdio.h"
main()
{ int i,len;
  double x[20],y[20];
  FILE *fp1, *fp2;
  len = 20;
  fp1 = fopen("x.dat","r");
  fp2 = fopen("y.dat","r");
  for (i=0;i<len;i++)
    fscanf(fp1,"%lf",&x[i]);
  printf("Original Sequence\n");
  for (i=0;i<len;i+=4)
    { printf("        %10.7f          %10.7f",x[i],x[i+1]);
      printf("        %10.7f          %10.7f\n",x[i+2],x[i+3]);
    }
  for (i=0;i<len;i++)
    fscanf(fp2,"%lf",&y[i]);
  printf("Linear Convolution\n");
  for (i=0;i<len;i+=4)
    { printf("        %10.7f          %10.7f",y[i],y[i+1]);
      printf("        %10.7f          %10.7f\n",y[i+2],y[i+3]);
    }
  fclose(fp1);
  fclose(fp2);
}
```

程序 convout.c 的运行结果:

序列 $x(i)$ $(i=0,1,\cdots,19)$为

```
1.0000000      0.9900500      0.9801990      0.9704460
0.9607890      0.9512290      0.9417650      0.9323940
```

0.9231160	0.9139310	0.9048370	0.8958340
0.8869200	0.8780950	0.8693580	0.8607080
0.8521440	0.8436650	0.8352700	0.8269590

线性卷积 $y(i)$ $(i=0,1,\cdots,19)$为

0.0000000	−1.0000000	−0.9900500	−0.9801990
−0.9704460	−0.9607890	−0.9512290	−0.9417650
−0.9323940	−0.9231160	−0.9139310	−0.9048370
−0.8958340	−0.8869200	−0.8780950	−0.8693580
−0.8607080	−0.8521440	−0.8436650	−0.8352700

§3.4 快速相关

一、功　能

用快速傅立叶变换计算两个有限长序列的线性相关。

二、方法简介

设序列 $x(n)$的长度为 M,序列 $y(n)$的长度为 N,序列 $x(n)$与 $y(n)$的互相关定义为

$$z(n)=\sum_{i=0}^{M-1}x(i)y(i+n)\ ,\quad n=-(M-1),\cdots,N-1$$

序列 $x(n)$的自相关定义为

$$z(n)=\sum_{i=0}^{M-1}x(i)x(i+n)\ ,\quad n=-(M-1),\cdots,M-1$$

显然,如果 $M=N$ 并且序列 $y(n)$与 $x(n)$相同,那么互相关就变成自相关。

用快速傅立叶变换计算线性相关的算法如下:

1. 选择 L 满足下述条件

$$\begin{cases}L\geqslant M+N-1\\ L=2^{\gamma},\quad \gamma\text{为正整数}\end{cases}$$

2. 将序列 $x(n)$与 $y(n)$按如下方式补零,形成长为 $L=2^{\gamma}$ 的序列

$$\bar{x}(n)=\begin{cases}x(n)\ , & n=0,1,\cdots,M-1\\ 0\ , & n=M,M+1,\cdots,L-1\end{cases}$$

$$\bar{y}(n)=\begin{cases}0\ , & n=0,1,\cdots,M-2\\ y(n)\ , & n=M-1,M,\cdots,M+N-2\\ 0\ , & n=M+N-1,M+N,\cdots,L-1\end{cases}$$

3. 用 FFT 算法分别计算序列 $\bar{x}(n)$与 $\bar{y}(n)$的离散傅立叶变换 $X(k)$与 $Y(k)$

$$X(k)=\sum_{n=0}^{L-1}\bar{x}(n)e^{-j2\pi nk/L}$$

$$Y(k)=\sum_{n=0}^{L-1}\bar{y}(n)e^{-j2\pi nk/L}$$

4. 计算 $X(k)$与 $Y(k)$的乘积

$$Z(k) = X^*(k)Y(k)$$

其中 × 表示复共轭；

5. 用 FFT 算法计算 $Z(k)$的离散傅立叶反变换，得到线性相关 $z(n)$

$$z(n) = \frac{1}{L}\sum_{k=0}^{L-1} Z(k)e^{j2\pi nk/L}, \quad n = 0,1,\cdots,L-1$$

序列 $z(n)$的前 $M+N-1$ 点的值就是序列 $x(n)$与 $y(n)$的线性相关。

三、使用说明

1. 子函数语句

void correl(x,y,m,n,len)

2. 形参说明

x —— 双精度实型一维数组，长度为 m。开始时存放实序列 $x(i)$，最后存放线性相关的值。

y —— 双精度实型一维数组，长度为 n。存放实序列 $y(i)$。

m —— 整型变量。序列 $x(i)$的长度。

n —— 整型变量。序列 $y(i)$的长度。

len —— 整型变量。len$\geqslant m+n-1$，且必须是 2 的整数次幂，即 len$=2^\gamma$。

四、子函数程序(文件名:correl. c)

```
#include "stdlib.h"
#include "rfft.c"
#include "irfft.c"
void correl(x,y,m,n,len)
int m,n,len;
double x[],y[];
{ int i,len2;
  double t, *z;
  z = malloc(len*sizeof(double));
  for (i=m;i<len;i++)
    { x[i] = 0.0; }
  for (i=0;i<(m-1);i++)
    { z[i] = 0.0; }
  for (i=(m-1);i<=(m+n-2);i++)
    { z[i] = y[i-m+1]; }
  for (i=(m+n-1);i<len;i++)
    { z[i] = 0.0; }
  rfft(x,len);
```

```
  rfft(z,len);
  len2 = len/2;
  x[0] = x[0]*z[0];
  x[len2] = x[len2]*z[len2];
  for (i=1;i<len2;i++)
    { t = x[i]*z[i] + x[len-i]*z[len-i];
      x[len-i] = x[i]*z[len-i] - x[len-i]*z[i];
      x[i] = t;
    }
  irfft(x,len);
  free(z);
}
```

五、例　题

设序列 $x(i)$ 为{−1,5,2,7,6,9,4}，其长度 $m=7$；序列 $y(i)$ 为{1,2,1,−3,4,5}，其长度 $n=6$。计算序列 $x(i)$与 $y(i)$的线性互相关以及 $x(i)$的自相关，并与直接计算结果进行比较。

主函数程序(文件名:correl. m)：

```
#include "correl.c"
main()
{ int i,j,m,n,m1,len,len1;
  double z[16];
  double x[16]={-1,5,2,7,6,9,4}, y[16]={1,2,1,-3,4,5};
  m = 7;
  n = 6;
  len = 16;
  len1 = m + n - 1;
  m1 = m-1;
  for (j= -m1;j<n;j++)
    { z[j+m1] = 0.0;
      for (i=0;i<m;i++)
        { if ( ( (i+j) >= 0 ) && ( (i+j) < n ) )
            z[j+m1] = z[j+m1] + x[i]*y[i+j];
        }
    }
  printf("Direct Calculation of Linear Cross-correlation\n");
  for (i=0;i<len1;i++)
    { printf("      %10.1lf",z[i]);
```

```
      if (i%4==3) printf("\n");
    }
  printf("\n");
  correl(x,y,m,n,len);
  printf("Fast Calculation of Linear Cross-correlation\n");
  for (i=0;i<len1;i++)
    { printf("        %10.1lf",x[i]);
      if (i%4==3) printf("\n");
    }
  printf("\n");
  m = 7;
  n = 7;
  len = 16;
  len1 = m + n - 1;
  m1 = m-1;
  for (j= -m1;j<n;j++)
    { z[j+m1] = 0.0;
      for (i=0;i<m;i++)
        { if ( ( (i+j) >= 0 ) && ( (i+j) < n ) )
            z[j+m1] = z[j+m1] + x[i] * x[i+j];
        }
    }
  printf("Direct Calculation of Linear Autocorrelation\n");
  for (i=0;i<len1;i++)
    { printf("        %10.1lf",z[i]);
      if (i%4==3) printf("\n");
    }
  printf("\n");
  correl(x,x,m,n,len);
  printf("Fast Calculation of Linear Autocorrelation\n");
  for (i=0;i<len1;i++)
    { printf("        %10.1lf",x[i]);
      if (i%4==3) printf("\n");
    }
  printf("\n");
}
```

运行结果：

序列 $x(i)$ 与 $y(i)$ 的线性互相关的直接计算结果

4.0	17.0	28.0	16.0
11.0	54.0	59.0	55.0
27.0	33.0	21.0	−5.0

序列 $x(i)$的线性互相关的快速算法结果

4.0	17.0	28.0	16.0
11.0	54.0	59.0	55.0
27.0	33.0	21.0	−5.0

序列 $x(i)$的线性自相关的直接计算结果

236.0	1219.0	2614.0	2707.0
2205.0	4948.0	7863.0	4948.0
2205.0	2707.0	2614.0	1219.0
236.0			

序列 $x(i)$的线性自相关的快速算法结果

236.0	1219.0	2614.0	2707.0
2205.0	4948.0	7863.0	4948.0
2205.0	2707.0	2614.0	1219.0
236.0			

第四章　数字滤波器的时域和频域响应

§4.1　数字滤波器的频率响应

一、功　能

计算数字滤波器的频率响应、幅频响应和相频响应。

二、方法简介

设数字滤波器的输入为 $x(k)$，输出为 $y(k)$，则该滤波器可用差分方程表示为

$$y(k) + \sum_{i=1}^{N} a(i)y(k-i) = \sum_{i=0}^{M} b(i)x(k-i)$$

其传递函数为

$$H(z) = \frac{b(0) + b(1)z^{-1} + \cdots + b(M)z^{-M}}{1 + a(1)z^{-1} + \cdots + a(N)z^{-N}}$$

其中 $a(i)$ $(i=1,2,\cdots,N)$ 和 $b(i)$ $(i=0,1,\cdots,M)$ 分别是数字滤波器分母多项式和分子多项式的系数。

数字滤波器的频率响应 $H(\omega)$ 为

$$H(\omega) = \frac{b(0) + b(1)e^{-j\omega} + \cdots + b(M)e^{-jM\omega}}{1 + a(1)e^{-j\omega} + \cdots + a(N)e^{-jN\omega}}$$

即

$$\begin{aligned} H(\omega) &= \mathrm{Re}[H(\omega)] + j\,\mathrm{Im}[H(\omega)] \\ &= |H(\omega)|e^{j\varphi(\omega)} \end{aligned}$$

其中 $|H(\omega)|$ 是数字滤波器的幅频响应，若用分贝表示，则为 $20\log_{10}|H(\omega)|$ (dB)。数字滤波器的相频响应可表示为

$$\varphi(\omega) = \tan^{-1}\left(\frac{\mathrm{Im}[H(\omega)]}{\mathrm{Re}[H(\omega)]}\right)$$

三、使用说明

1. 子函数语句

void gain(b,a,m,n,x,y,len,sign)

2. 形参说明

b ——双精度实型一维数组，长度为(m+1)。存放滤波器分子多项式的系数 $b(i)$。

a ——双精度实型一维数组，长度为(n+1)。存放滤波器分母多项式的系数 $a(i)$。

m ——整型变量。滤波器分子多项式的阶数。

n ——整型变量。滤波器分母多项式的阶数。

x ——双精度实型一维数组，长度为 len。当 sign=0 时，存放滤波器频率响应的实部 Re[$H(\omega)$]；当 sign=1 时，存放滤波器幅频响应$|H(\omega)|$；当 sign=2 时，存放用分贝表示的滤波器幅频响应$|H(\omega)|$。

y ——双精度实型一维数组，长度为 len。当 sign=0 时，存放滤波器频率响应的虚部 Im[$H(\omega)$]；当 sign=1 和 2 时，存放滤波器的相频响应 $\varphi(\omega)$。

len——整型变量。频率响应的长度。

sign ——整型变量。当 sign=0 时，计算滤波器频率响应的实部 Re[$H(\omega)$]和虚部 Im[$H(\omega)$]；当 sign=1 时，计算滤波器的幅频响应$|H(\omega)|$和相频响应 $\varphi(\omega)$；当 sign=2 时，计算滤波器的幅频响应$|H(\omega)|$（用 dB 表示）和相频响应 $\varphi(\omega)$。

四、子函数程序（文件名：gain.c）

```
#include "math.h"
void gain(b,a,m,n,x,y,len,sign)
int m,n,len,sign;
double b[], a[], x[], y[];
{ int i,k;
  double ar,ai,br,bi,zr,zi,im,re,den,numr,numi,freq,temp;
  for (k=0;k<len;k++)
    { freq = k * 0.5/(len-1);
      zr = cos(-8.0 * atan(1.0) * freq);
      zi = sin(-8.0 * atan(1.0) * freq);
      br = 0.0;
      bi = 0.0;
      for (i=m;i>0;i--)
        { re = br;
          im = bi;
          br = ( re + b[i] ) * zr - im * zi;
          bi = ( re + b[i] ) * zi + im * zr;
        }
      ar = 0.0;
      ai = 0.0;
      for (i=n;i>0;i--)
        { re = ar;
          im = ai;
          ar = ( re + a[i] ) * zr - im * zi;
          ai = ( re + a[i] ) * zi + im * zr;
        }
```

```
        br = br + b[0];
        ar = ar + 1.0;
        numr = ar * br + ai * bi;
        numi = ar * bi - ai * br;
        den = ar * ar + ai * ai;
        x[k] = numr/den;
        y[k] = numi/den;
        switch (sign)
          { case 1:
              { temp = sqrt(x[k] * x[k] + y[k] * y[k]);
                y[k] = atan2(y[k],x[k]);
                x[k] = temp;
                break;
              }
            case 2:
              { temp = x[k] * x[k] + y[k] * y[k];
                y[k] = atan2(y[k],x[k]);
                x[k] = 10.0 * log10(temp);
              }
          }
      }
  }
```

五、例　题

数字系统的传递函数为

$$H(z) = \frac{-0.1z^{-1}}{1 + 0.9z^{-2}}$$

求该系统的幅频响应和相频响应，并画出相应的图形。

主函数程序(文件名:gain.m):

```
#include "stdio.h"
#include "gain.c"
main()
{ int i;
  double a[]={1.0, 0.0, 0.9}, b[]={0.0, -0.1};
  double f,x[300],y[300];
  FILE *fp;
  gain(b,a,1,2,x,y,300,1);
```

```
    if ( ( fp = fopen("gainam.dat","w")) == NULL )
      { printf("cannot open file 'gainam.dat' ! \n");
        exit(0);
      }
    for(i=0;i<300;i++)
      { f=i * 0.5/299;
        fprintf(fp," %lf %lf\n",f,x[i]); }
    fclose(fp);
    if ( ( fp = fopen("gainph.dat","w")) == NULL )
      { printf("cannot open file 'gainph.dat' ! \n");
        exit(0);
      }
    for(i=0;i<300;i++)
      { f=i * 0.5/299;
        fprintf(fp," %lf %lf\n",f,y[i]); }
    fclose(fp);
}
```

运行结果：

该数字系统的幅频响应见图2－4－1a)，相频响应见图2－4－1b)。从这两个图可以看出，系统的幅频响应在频率为0.25处有一个谐振峰，其相位在该处从π到0有一个跃变。

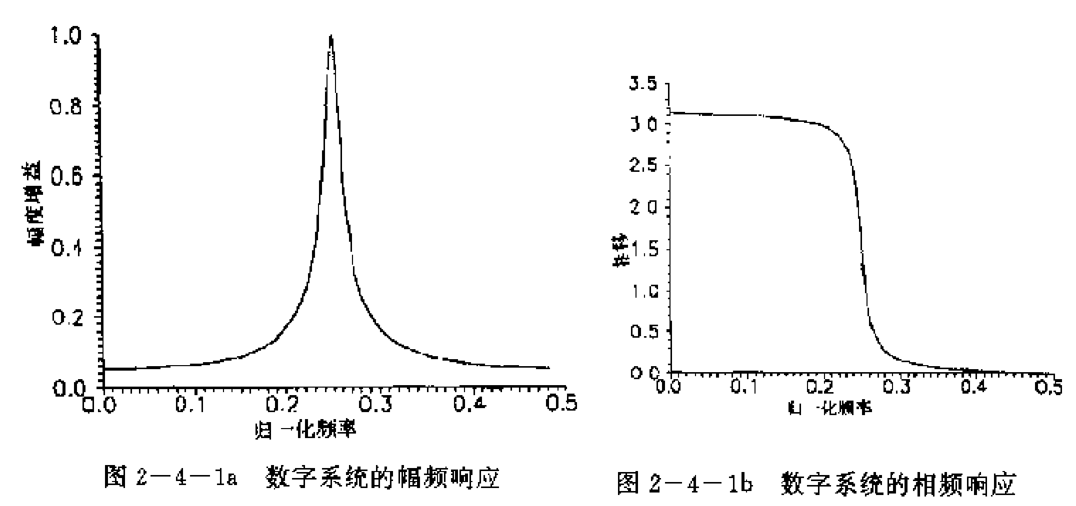


图2－4－1a　数字系统的幅频响应　　　图2－4－1b　数字系统的相频响应

§4.2　级联型数字滤波器的频率响应

一、功　能

计算级联型数字滤波器的频率响应、幅频响应和相频响应。

二、方法简介

级联型数字滤波器的传递函数为

$$H(z) = \prod_{j=1}^{L} H_j(z) = \prod_{j=1}^{L} \frac{b(j,0) + b(j,1)z^{-1} + \cdots + b(j,N)z^{-N}}{1 + a(j,1)z^{-1} + \cdots + a(j,N)z^{-N}}$$

其中 N 阶节 $H_j(z)$ 为

$$H_j(z) = \frac{b(j,0) + b(j,1)z^{-1} + \cdots + b(j,N)z^{-N}}{1 + a(j,1)z^{-1} + \cdots + a(j,N)z^{-N}}$$

这里 $a(j,i)$ 和 $b(j,i)$ $(j=1,2,\cdots,L;\ i=0,1,\cdots,N)$ 分别是数字滤波器分母多项式和分子多项式的系数。

级联型数字滤波器的频率响应 $H(\omega)$ 为

$$H(\omega) = \prod_{j=1}^{L} \frac{b(j,0) + b(j,1)e^{-j\omega} + \cdots + b(j,N)e^{-jN\omega}}{1 + a(j,1)e^{-j\omega} + \cdots + a(j,N)e^{-jN\omega}}$$

即

$$H(\omega) = \mathrm{Re}[H(\omega)] + j\,\mathrm{Im}[H(\omega)] = |H(\omega)| e^{j\varphi(\omega)}$$

其中 $|H(\omega)|$ 是数字滤波器的幅频响应，若用分贝表示，则为 $20\log_{10}|H(\omega)|$ (dB)。数字滤波器的相频响应可表示为

$$\varphi(\omega) = \tan^{-1}\left(\frac{\mathrm{Im}[H(\omega)]}{\mathrm{Re}[H(\omega)]}\right)$$

三、使用说明

1. 子函数语句

void gainc(b,a,n,ns,x,y,len,sign)

2. 形参说明

b ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分子多项式的系数，b[j][i] 表示第 j 个 n 阶节的分子多项式的第 i 个系数。

a ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分母多项式的系数，a[j][i]表示第 j 个 n 阶节的分母多项式的第 i 个系数。

n ——整型变量。级联型滤波器每节的阶数。

ns ——整型变量。级联型滤波器的 n 阶节数 L。

x ——双精度实型一维数组，长度为 len。当 sign=0 时，存放滤波器频率响应的实部 Re $[H(\omega)]$；当 sign=1 时，存放滤波器幅频响应 $|H(\omega)|$；当 sign=2 时，存放用分贝表示的滤波器幅频响应 $|H(\omega)|$。

y ——双精度实型一维数组，长度为 len。当 sign=0 时，存放滤波器频率响应的虚部 Im $[H(\omega)]$；当 sign=1 和 2 时，存放滤波器的相频响应 $\varphi(\omega)$。

len——整型变量。频率响应的长度。

sign ——整型变量。当 sign=0 时，计算滤波器频率响应的实部 Re$[H(\omega)]$和虚部 Im $[H(\omega)]$；当 sign=1 时，计算滤波器的幅频响应 $|H(\omega)|$ 和相频响应 $\varphi(\omega)$；当 sign=2 时，计算滤波器的幅频响应 $|H(\omega)|$（用 dB 表示）和相频响应 $\varphi(\omega)$。

四、子函数程序（文件名：gainc.c）

```
#include "math.h"
void gainc(b,a,n,ns,x,y,len,sign)
int n,ns,len,sign;
double b[], a[], x[], y[];
{ int i,j,k,n1;
  double ar,ai,br,bi,zr,zi,im,re,den,numr,numi,freq,temp;
  double hr,hi,tr,ti;
  n1 = n + 1;
  for (k=0;k<len;k++)
    { freq = k * 0.5/(len-1);
      zr = cos(-8.0 * atan(1.0) * freq);
      zi = sin(-8.0 * atan(1.0) * freq);
      x[k]=1.0;
      y[k]=0.0;
      for (j=0;j<ns;j++)
        { br = 0.0;
          bi = 0.0;
          for (i=n;i>0;i--)
            { re = br;
              im = bi;
```

```
          br = ( re + b[j * n1+i] ) * zr - im * zi;
          bi = ( re + b[j * n1+i] ) * zi + im * zr;
        }
      ar = 0.0;
      ai = 0.0;
      for (i=n;i>0;i--)
        { re = ar;
          im = ai;
          ar = ( re + a[j * n1+i] ) * zr - im * zi;
          ai = ( re + a[j * n1+i] ) * zi + im * zr;
        }
      br = br + b[j * n1+0];
      ar = ar + 1.0;
      numr = ar * br + ai * bi;
      numi = ar * bi - ai * br;
      den = ar * ar + ai * ai;
      hr = numr/den;
      hi = numi/den;
      tr = x[k] * hr - y[k] * hi;
      ti = x[k] * hi + y[k] * hr;
      x[k] = tr;
      y[k] = ti;
    }
  switch (sign)
    { case 1:
        { temp = sqrt(x[k] * x[k] + y[k] * y[k]);
          if (temp != 0.0)
            { y[k] = atan2(y[k],x[k]); }
          else
            { y[k] = 0.0; }
          x[k] = temp;
          break;
        }
      case 2:
        { temp = x[k] * x[k] + y[k] * y[k];
          if (temp != 0.0)
            { y[k] = atan2(y[k],x[k]); }
          else
```

```
        { temp = 1.0e-40;
          y[k] = 0.0;
        }
        x[k] = 10.0 * log10(temp);
      }
    }
  }
}
```

五、例　题

数字滤波器由两个子系统 $H_1(z)$和 $H_2(z)$级联而成，其中 $H_1(z)$为

$$H_1(z) = \frac{0.2}{1 - 0.8z^{-1}}$$

$H_2(z)$为

$$H_2(z) = \frac{0.05 - 0.1z^{-1}}{1 - 0.95z^{-1} + 0.9z^{-2}}$$

选取参数 n=2，ns=2，len=300，求该滤波器的幅频响应。

主函数程序(文件名:gainc.m)：

```
#include "stdio.h"
#include "gainc.c"
main()
{ int i,n,ns;
  double a[2][3]={{1.0, -0.8, 0.0},{1.0, -0.95, 0.9}};
  double b[2][3]={{0.2, 0.0, 0.0},{0.05, -0.1, 0.0}};
  double f,x[300],y[300];
  FILE *fp;
  n = 2;
  ns = 2;
  gainc(b,a,n,ns,x,y,300,1);
  if ( ( fp = fopen("gaincam.dat","w")) == NULL )
    { printf("cannot open file 'gaincam.dat' ! \n");
      exit(0);
    }
  for(i=0;i<300;i++)
    { f=i*0.5/299;
      fprintf(fp,"%lf   %lf\n",f,x[i]); }
  fclose(fp);
}
```

运行结果：

级联型数字滤波器的幅频响应如图 2—4—2 所示。

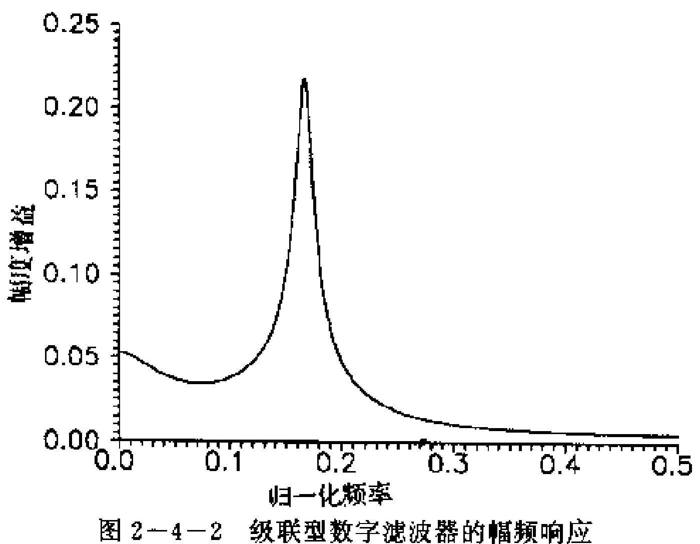


图 2—4—2　级联型数字滤波器的幅频响应

§4.3　数字滤波器的时域响应

一、功　能

计算数字滤波器的单位冲激响应和单位阶跃响应。

二、方法简介

设数字滤波器的输入为 $x(k)$，输出为 $y(k)$，则该滤波器可用差分方程表示为

$$y(k)+\sum_{i=1}^{N}a(i)\,y(k-i)=\sum_{i=0}^{M}b(i)\,x(k-i)$$

其传递函数为

$$H(z)=\frac{b(0)+b(1)z^{-1}+\cdots+b(M)z^{-M}}{1+a(1)z^{-1}+\cdots+a(N)z^{-N}}$$

其中 $a(i)$ $(i=1,2,\cdots,N)$ 和 $b(i)$ $(i=0,1,\cdots,M)$ 分别是数字滤波器分母多项式和分子多项式的系数。

当输入序列 $x(k)=\delta(k)$ 时，输出 $y(k)$ 就是滤波器的单位冲激响应；当输入序列 $x(k)=u(k)$ 时，输出 $y(k)$ 就是滤波器的单位阶跃响应；

三、使用说明

1. 子函数语句

```
void resp(x,y,lx,ly,b,a,m,n)
```

2. 形参说明

x ——双精度实型一维数组，长度为 lx。存放滤波器的输入序列。

y ——双精度实型一维数组，长度为 ly。存放滤波器的输出序列。

lx ——整型变量。输入序列的长度。

ly ——整型变量。输出序列的长度。

b ——双精度实型一维数组，长度为(m+1)。存放滤波器分子多项式的系数 $b(i)$。

a ——双精度实型一维数组，长度为(n+1)。存放滤波器分母多项式的系数 $a(i)$。

m ——整型变量。滤波器分子多项式的阶数。

n ——整型变量。滤波器分母多项式的阶数。

四、子函数程序(文件名：resp. c)

```
void resp(x,y,lx,ly,b,a,m,n)
int lx,ly,m,n;
double x[],y[],b[],a[];
{ int k,i,i1;
  double sum;
  for (k=0;k<ly;k++)
    { sum = 0.0;
      for(i=0;i<=m;i++)
        { if ( (k-i) >= 0 )
            { i1 = ( (k-i) < lx ) ? (k-i) : (lx-1);
              sum = sum + b[i] * x[i1];
            }
        }
      for (i=1;i<=n;i++)
        { if ( (k-i) >= 0 )
            sum = sum - a[i] * y[k-i];
        }
      y[k] = sum;
    }
}
```

五、例　题

数字滤波器由两个子系统 $H_1(z)$和 $H_2(z)$级联而成，其中 $H_1(z)$为

$$H_1(z) = \frac{0.05 - 0.1z^{-1}}{1 - 0.95z^{-1} + 0.9z^{-2}}$$

$H_2(z)$ 为

$$H_2(z) = \frac{0.2}{1 - 0.8z^{-1}}$$

选取参数 lx=2，ly=100，试求该滤波器的单位阶跃响应。

主函数程序(文件名:resp.m):

```
#include "stdio.h"
#include "resp.c"
main()
{ int i,m1,m2,n1,n2,lx,ly;
  double y[100],z[100];
  double x[2]={1.0, 1.0}, b[2]={0.05, -0.1}, a[3]={1.0,-0.95, 0.9};
  double d[1]={0.2}, c[2]={1.0, -0.8};
  FILE *fp;
  lx = 2;
  ly = 100;
  m1 = 1; n1 = 2;
  m2 = 0; n2 = 1;
  if ( ( fp = fopen("resp.dat","w")) == NULL )
    { printf("can not open file 'resp.dat' ! \n");
      exit(0);
    }
  resp(x,y,lx,ly,b,a,m1,n1);
  resp(y,z,ly,ly,d,c,m2,n2);
  for(i=0;i<ly;i++)
    { fprintf(fp," %4d %lf\n",i,z[i]); }
  fclose(fp);
  for (i=0;i<32;i++)
    { printf(" %10.7lf",z[i]);
      if (i%4==3) printf("\n");
    }
}
```

运行结果:

单位阶跃响应(前 32 点)为

```
 0.0100000   0.0075000  -0.0134750  -0.0388312
-0.0501862  -0.0430680  -0.0300183  -0.0271734
-0.0387318  -0.0542862  -0.0602707  -0.0532457
-0.0422166  -0.0388861  -0.0463080  -0.0568841
-0.0606738  -0.0550934  -0.0466514  -0.0438701
-0.0489986  -0.0565122  -0.0591451  -0.0549727
-0.0487101  -0.0465725  -0.0502234  -0.0556519
-0.0575522  -0.0544950  -0.0498990  -0.0482991
```

数字滤波器的单位阶跃响应如图 2—4—3 所示。

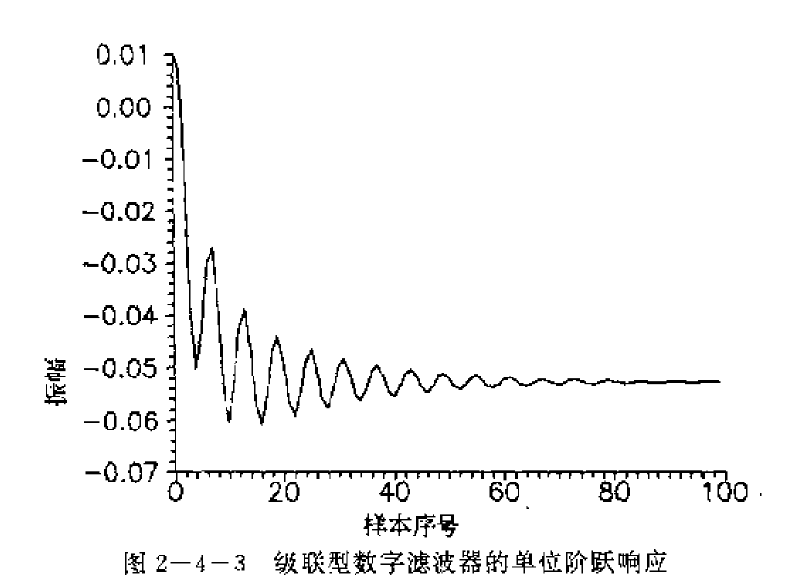


图 2—4—3　级联型数字滤波器的单位阶跃响应

§4.4　直接型 IIR 数字滤波(一)

一、功　能

用直接型 IIR 数字滤波器进行数字滤波。

二、方法简介

设 IIR 数字滤波器的输入为 $x(k)$，输出为 $y(k)$，则该滤波器可用差分方程表示为

$$y(k) + \sum_{i=1}^{N} a(i)y(k-i) = \sum_{i=0}^{M} b(i)x(k-i)$$

其传递函数为

$$H(z) = \frac{b(0) + b(1)z^{-1} + \cdots + b(M)z^{-M}}{1 + a(1)z^{-1} + \cdots + a(N)z^{-N}}$$

其中 $a(i)$ $(i=1,2,\cdots,N)$ 和 $b(i)$ $(i=0,1,\cdots,M)$ 分别是数字滤波器分母多项式和分子多项式的系数。

当给定数字滤波器系数 $a(i)$ 与 $b(i)$ 后，即可用上式对输入序列 $x(k)$ 进行数字滤波，从而得到滤波器的输出 $y(k)$

三、使用说明

1. 子函数语句

```
void filter(b,a,m,n,x,len,px,py)
```

2. 形参说明

b ——双精度实型一维数组，长度为(m+1)。存放滤波器分子多项式的系数 $b(i)$。

a ——双精度实型一维数组，长度为(n+1)。存放滤波器分母多项式的系数 $a(i)$。

m ——整型变量。滤波器分子多项式的阶数。

n ——整型变量。滤波器分母多项式的阶数。

x ——双精度实型一维数组，长度为len。开始时存放滤波器的输入序列，最后存放滤波器的输出序列；在分块处理时，它用于表示当前块内的滤波器的输入序列与输出序列。

len——整型变量。输入序列与输出序列的长度；在分块处理时，它用于表示块的长度。

px ——双精度实型一维数组，长度为(m+1)。在分块处理时，它用于保存前一块滤波时的(m+1)个输入序列值。

py ——双精度实型一维数组，长度为(n+1)。在分块处理时，它用于保存前一块滤波时的 n 个输出序列值。

当输入序列 x(k)很长时，由于计算机内存的限制，常将其分成彼此衔接的若干块进行处理。数组 px 与 py 就是专为分块处理而设置的。px 用于保存前一块滤波时的(m+1)个输入序列值，即 px[ ]={x(k), x(k−1), …, x(k−m)};py 用于保存前一块滤波时的 n 个输出序列值，即 py[ ]={y(k−1), y(k−2), …, y(k−n)}。通常，我们假定滤波器的初始条件为零，因此数组 px[ ]与 py[ ]在滤波前都要初始化为零。有关分块处理的具体方法见后面的例题。

四、子函数程序(文件名：filter.c)

```
#include "math.h"
void filter(b,a,m,n,x,len,px,py)
int m,n,len;
double a[],b[],x[],px[],py[];
{ int k,i;
  for (k=0;k<len;k++)
    { px[0] = x[k];
      x[k] = 0.0;
      for (i=0;i<=m;i++)
        { x[k] = x[k] + b[i] * px[i]; }
      for (i=1;i<=n;i++)
        { x[k] = x[k] - a[i] * py[i]; }
      if ( fabs(x[k] ) > 1.0e10 )
        { printf("This is an unstable filter! \n");
          exit(0);
        }
      for (i=m;i>=1;i--)
        { px[i] = px[i-1]; }
      for (i=n;i>=2;i--)
        { py[i] = py[i-1]; }
```

```
        py[1] = x[k];
      }
  }
```

五、例 题

4 阶切比雪夫低通数字滤波器的传递函数为

$$H(z) = \frac{0.001836 + 0.007344z^{-1} + 0.011016z^{-2} + 0.007344z^{-3} + 0.001836z^{-4}}{1.0 - 3.0544z^{-1} + 3.8291z^{-2} - 2.2925z^{-3} + 0.55075z^{-4}}$$

选取参数 m=4，n=4，块长 len=25，求该滤波器的单位冲激响应，并画出相应的图形。

主函数程序(文件名:filter.m)：

```
  #include "stdio.h"
  #include "filter.c"
  main()
  { int i,k,m,n,len,nblk;
    double b[5]={0.001836, 0.007344, 0.011016, 0.007344, 0.001836};
    double a[5]={1.0, -3.0544, 3.8291, -2.2925, 0.55075};
    double px[5]={0.,0.,0.,0.,0.};
    double py[5]={0.,0.,0.,0.,0.};
    double x[25],data[100];
    FILE *fp;
    m = 4;
    n = 4;
    len = 25;
    nblk = 4;
    x[0] = 1.0;
    for (k=1;k<len;k++)
      { x[k] = 0.0; }
    for (i=0; i<nblk; i++)
      { filter(b,a,m,n,x,len,px,py);
        for ( k=0; k<len; k++)
          { data[i*len+k] = x[k];
            x[k] = 0.0;
          }
      }
    printf("Unit Impulse Response\n");
    for (i=0;i<16;i++)
      { printf(" %10.7lf",data[i]);
        if ( i%4==3 ) printf("\n");
```

```
    }
  if ( ( fp = fopen("filter.dat","w")) == NULL )
    { printf("cannot open file 'filter.dat' ! \n");
      exit(0);
    }
  for (i=0;i<100;i++)
    { fprintf(fp,"%2d %lf\n",i,data[i]); }
  fclose(fp);
}
```

运行结果：

单位冲激响应(前 16 点)为

0.0018360	0.0129519	0.0435460	0.0949659
0.1538388	0.1989473	0.2123269	0.1871151
0.1298634	0.0573615	−0.0100326	−0.0556283
−0.0715163	−0.0600246	−0.0314986	0.0003172

单位冲激响应如图 2—4—4 所示。

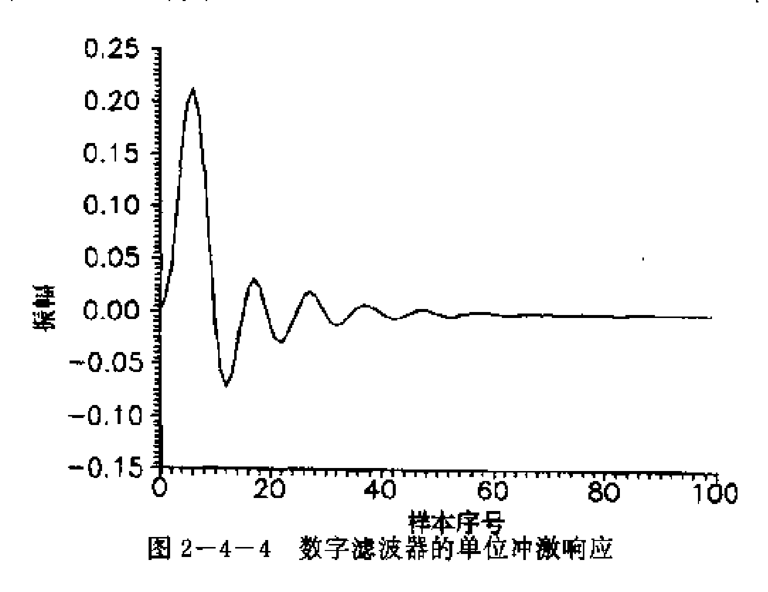


图 2—4—4　数字滤波器的单位冲激响应

§4.5　直接型 IIR 数字滤波(二)

一、功　能

用直接型 IIR 数字滤波器进行数字滤波。

二、方法简介

设 IIR 数字滤波器的输入为 $x(k)$，输出为 $y(k)$，则该滤波器可用差分方程表示为

$$y(k)+\sum_{i=1}^{N}a(i)y(k-i)=\sum_{i=0}^{M}b(i)x(k-i)$$

其传递函数为

$$H(z)=\frac{b(0)+b(1)z^{-1}+\cdots+b(M)z^{-M}}{1+a(1)z^{-1}+\cdots+a(N)z^{-N}}$$

其中 $a(i)$ $(i=1,2,\cdots,N)$和 $b(i)$ $(i=0,1,\cdots,M)$分别是数字滤波器分母多项式和分子多项式的系数。

当给定数字滤波器系数 $a(i)$与 $b(i)$后，即可用上式对输入序列 $x(k)$进行数字滤波，从而得到滤波器的输出 $y(k)$。

三、使用说明

1. 子函数语句

void filtery(b,a,m,n,x,y,len,px,py)

2. 形参说明

b ——双精度实型一维数组，长度为(m+1)。存放滤波器分子多项式的系数 $b(i)$。

a ——双精度实型一维数组，长度为(n+1)。存放滤波器分母多项式的系数 $a(i)$。

m ——整型变量。滤波器分子多项式的阶数。

n ——整型变量。滤波器分母多项式的阶数。

x ——双精度实型一维数组，长度为 len。存放滤波器的输入序列；在分块处理时，它用于表示当前块内的滤波器的输入序列。

y ——双精度实型一维数组，长度为 len。存放滤波器的输出序列；在分块处理时，它用于表示当前块内的滤波器的输出序列。在滤波前必须将其初始化为零。

len——整型变量。输入序列与输出序列的长度；在分块处理时，它用于表示块的长度。

px ——双精度实型一维数组，长度为(m+1)。在分块处理时，它用于保存前一块滤波时的(m+1)个输入序列值。

py ——双精度实型一维数组，长度为(n+1)。在分块处理时，它用于保存前一块滤波时的 n 个输出序列值。

当输入序列 x(k)很长时，由于计算机内存的限制，常将其分成彼此衔接的若干块进行处理。数组 px 与 py 就是专为分块处理而设置的。px 用于保存前一块滤波时的(m+1)个输入序列值，即 px[]={x(k), x(k−1), …, x(k−m)}；py 用于保存前一块滤波时的 n 个输出序列值，即 py[]={y(k−1), y(k−2), …, y(k−n)}。通常，我们假定滤波器的初始条件为零，因此，数组 px[]与 py[]在滤波前都要初始化为零。

四、子函数程序(文件名：filtery.c)

```
#include "math.h"
void filtery(b,a,m,n,x,y,len,px,py)
int m,n,len;
```

```
double a[],b[],x[],y[],px[],py[];
{ int k,i;
  double sum;
  for (k=0;k<len;k++)
    { px[0] = x[k];
      sum = 0.0;
      for (i=0;i<=m;i++)
        { sum = sum + b[i] * px[i]; }
      for (i=1;i<=n;i++)
        { sum = sum - a[i] * py[i]; }
      if ( fabs(x[k] ) > 1.0e10 )
        { printf("This is an unstable filter! \n");
          exit(0);
        }
      for (i=m;i>=1;i--)
        { px[i] = px[i-1]; }
      for (i=n;i>=2;i--)
        { py[i] = py[i-1]; }
      py[1] = sum;
      y[k] = y[k] + sum;
    }
}
```

五、例　题

4 阶切比雪夫低通数字滤波器的传递函数为

$$H(z) = \frac{0.001836 + 0.007344z^{-1} + 0.011016z^{-2} + 0.007344z^{-3} + 0.001836z^{-4}}{1.0 - 3.0544z^{-1} + 3.8291z^{-2} - 2.2925z^{-3} + 0.55075z^{-4}}$$

选取参数 m=4，n=4，len=100，求该滤波器的单位冲激响应，并画出相应的图形。

主函数程序(文件名:filtery.m)：

```
#include "stdio.h"
#include "filtery.c"
main()
{ int i,m,n,len;
  double b[5]={0.001836, 0.007344, 0.011016, 0.007344, 0.001836};
  double a[5]={1.0, -3.0544, 3.8291, -2.2925, 0.55075};
  double px[5]={0.,0.,0.,0.,0.};
  double py[5]={0.,0.,0.,0.,0.};
  double x[100],y[100];
```

```
    FILE *fp;
    m = 4;
    n = 4;
    len = 100;
    for (i=0;i<len;i++)
      { x[i] = 0.0;
        y[i] = 0.0;
      }
    x[0] = 1.0;
    filtery(b,a,m,n,x,y,len,px,py);
    printf("Unit Impulse Response\n");
    for (i=0;i<16;i++)
      { printf(" %10.7lf",y[i]);
        if ( i%4==3 ) printf("\n");
      }
    if ( ( fp = fopen("filtery.dat","w")) == NULL )
      { printf("cannot open file 'filtery.dat' ! \n");
        exit(0);
      }
    for (i=0;i<100;i++)
      { fprintf(fp," %2d    %lf\n",i,y[i]); }
    fclose(fp);
  }
```

运行结果：

单位冲激响应(前 16 点)为

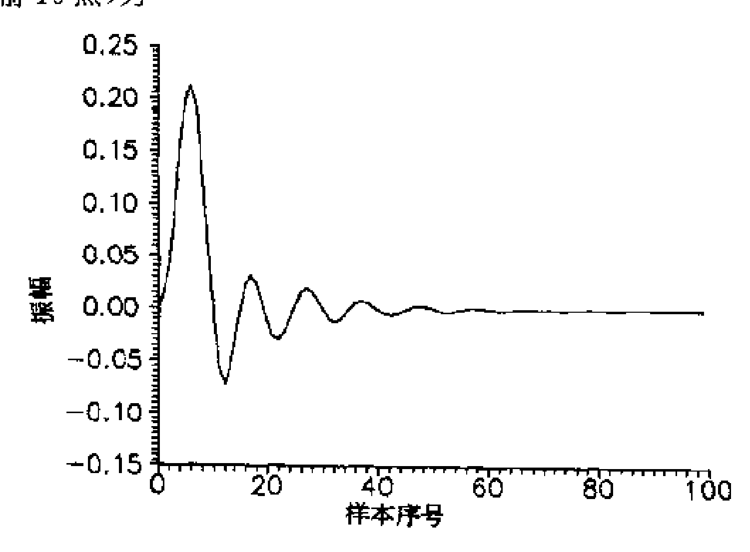


图 2—4—5　数字滤波器的单位冲激响应

0.0018360	0.0129519	0.0435460	0.0949659
0.1538388	0.1989473	0.2123269	0.1871151
0.1298634	0.0573615	−0.0100326	−0.0556283
−0.0715163	−0.0600246	−0.0314986	0.0003172

单位冲激响应如图 2—4—5 所示。

§4.6 级联型 IIR 数字滤波

一、功　能

用级联型 IIR 数字滤波器进行数字滤波。

二、方法简介

级联型 IIR 数字滤波器的传递函数为

$$H(z)=\prod_{j=1}^{L}H_j(z)$$

$$=\prod_{j=1}^{L}\frac{b(j,0)+b(j,1)z^{-1}+\cdots+b(j,N)z^{-N}}{1+a(j,1)z^{-1}+\cdots+a(j,N)z^{-N}}$$

其中 N 阶节 $H_i(z)$为

$$H_j(z)=\frac{b(j,0)+b(j,1)z^{-1}+\cdots+b(j,N)z^{-N}}{1+a(j,1)z^{-1}+\cdots+a(j,N)z^{-N}}$$

这里 $a(j,i)$和 $b(j,i)$ ($j=1,2,\cdots,L$; $i=0,1,\cdots,N$)分别是数字滤波器分母多项式和分子多项式的系数。

当给定数字滤波器系数 $a(j,i)$与 $b(j,i)$后,即可用上式对输入序列 $x(k)$进行数字滤波,从而得到滤波器的输出 $y(k)$

三、使用说明

1. 子函数语句

void filterc(b,a,n,ns,x,len,px,py)

2. 形参说明

b ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分子多项式的系数,b[j][i] 表示第 j 个 n 阶节的分子多项式的第 i 个系数。

a ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分母多项式的系数。a[j][i]表示第 j 个 n 阶节的分母多项式的第 i 个系数。

n ——整型变量。滤波器每节的阶数。

ns ——整型变量。滤波器的 n 阶节数：L。

x ——双精度实型一维数组，长度为len。开始时存放滤波器的输入序列，最后存放滤波器的输出序列；在分块处理时，它用于表示当前块内的滤波器的输入序列与输出序列。

len——整型变量。输入序列与输出序列的长度；在分块处理时，它用于表示块的长度

px ——双精度实型二维数组，体积为ns×(n+1)。在分块处理时，它用于保存前一块滤波时的(n+1)个输入序列值。

py ——双精度实型二维数组，体积为ns×(n+1)。在分块处理时，它用于保存前一块滤波时的n个输出序列值。

当输入序列x(k)很长时，由于计算机内存的限制，常将其分成彼此衔接的若干块进行处理。数组px与py就是专为分块处理而设置的。px用于保存前一块滤波时的(n+1)个输入序列值，即px[j][ ]={x(k),x(k−1),…,x(k−n)}；py用于保存前一块滤波时的n个输出序列值，即py[j][ ]={y(k−1),y(k−2),…,y(k−n)}。通常，我们假定滤波器的初始条件为零，因此数组px[ ][ ]与py[ ][ ]在滤波前都要初始化为零。有关分块处理的具体方法见后面的例题。

四、子函数程序(文件名：filterc.c)

```
#include "math.h"
void filterc(b,a,n,ns,x,len,px,py)
int n,ns,len;
double a[],b[],x[],px[],py[];
{ int i,j,k,n1;
  n1 = n + 1;
  for (j=0;j<ns;j++)
    { for (k=0;k<len;k++)
        { px[j*n1+0] = x[k];
          x[k] = b[j*n1+0] * px[j*n1+0];
          for (i=1;i<=n;i++)
          {x[k] += b[j*n1+i]*px[j*n1+i]-a[j*n1+i]*py[j*n1+i];}
          if ( fabs(x[k] ) > 1.0e10 )
            { printf("This is an unstable filter ! \n");
              exit(0);
            }
          for (i=n;i>=2;i--)
            { px[j*n1+i] = px[j*n1+i-1];
              py[j*n1+i] = py[j*n1+i-1];
            }
          px[j*n1+1] = px[j*n1+0];
```

```
            py[j * n]+1] = x[k];
          }
      }
}
```

五、例　题

4 阶切比雪夫低通数字滤波器的传递函数为

$$H(z)=\frac{0.001836(1+z^{-1})^4}{(1-1.4996z^{-1}+0.8482z^{-2})(1-1.5548z^{-1}+0.6493z^{-2})}$$

它由两个 2 阶节级联而成。选取参数 n=2，ns=2，共分 4 块进行处理，每块长度 len=25，求该滤波器的单位冲激响应，并画出相应的图形。

主函数程序(文件名:filterc.m)：

```
#include "stdio.h"
#include "filterc.c"
main()
{ int i,j,n,ns,len,nblk;
  double b[2][3]={{0.001836, 0.003672, 0.001836}, {1.0, 2.0, 1.0}};
  double a[2][3]={{1.0, -1.4996, 0.8482}, {1.0, -1.5548, 0.6493}};
  double px[2][3],py[2][3];
  double x[25],data[100];
  FILE *fp;
  n = 2;
  ns = 2;
  len = 25;
  nblk = 4;
  x[0] = 1.0;
  for (i=1;i<len;i++ )
    { x[i]=0.0; }
  for (i=0;i<ns;i++)
  for (j=0;j<=n;j++)
    { px[i][j]=0.0;
      py[i][j]=0.0;
    }
  for (j=0;j<nblk;j++)
    { filterc(b,a,n,ns,x,len,px,py);
      for (i=0;i<len;i++)
        { data[j*len+i] = x[i];
          x[i] = 0.0;
```

```
            }
        }
    printf("Unit Impulse Response\n");
    for (i=0;i<16;i++)
      { printf(" %10.7lf",data[i]);
        if ( i%4==3 ) printf("\n");
      }
    if ( ( fp = fopen("filterc.dat","w")) == NULL )
      { printf("cannot open file 'filterc.dat' ! \n");
        exit(0);
      }
    for (i=0;i<100;i++)
      { fprintf(fp,"%2d   %lf\n",i,data[i]); }
    fclose(fp);
  }
```

运行结果：

单位冲激响应(前 16 点)为

0.0018360	0.0129519	0.0435460	0.0949662
0.1538403	0.1989518	0.2123368	0.1871326
0.1298898	0.0573960	−0.0099931	−0.0555889
−0.0714825	−0.0600010	−0.0314873	0.0003168

单位冲激响应如图 2－4－6 所示。

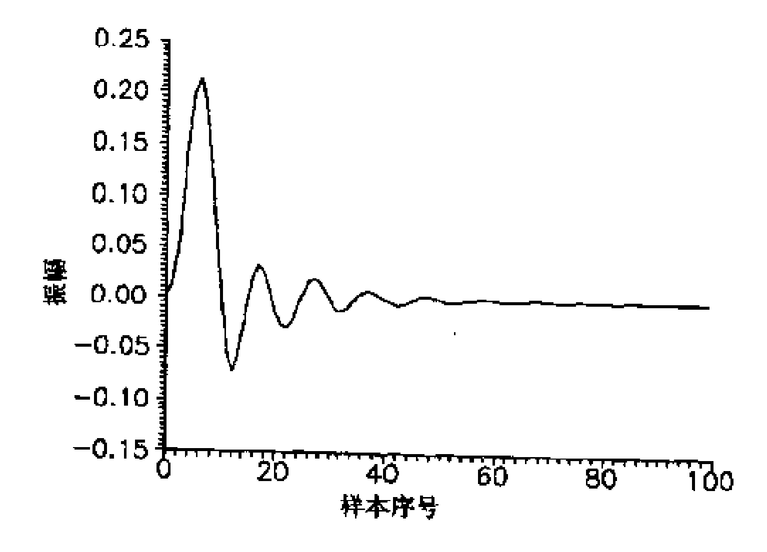


图 2－4－6　级联型数字滤波器的单位冲激响应

§4.7 并联型 IIR 数字滤波

一、功 能

用并联型 IIR 数字滤波器进行数字滤波。

二、方法简介

并联型 IIR 数字滤波器的传递函数为

$$H(z) = \sum_{j=1}^{L} H_j(z)$$

$$= \sum_{j=1}^{L} \frac{b(j,0) + b(j,1)z^{-1} + \cdots + b(j,N)z^{-N}}{1 + a(j,1)z^{-1} + \cdots + a(j,N)z^{-N}}$$

其中 N 阶节 $H_j(z)$ 为

$$H_j(z) = \frac{b(j,0) + b(j,1)z^{-1} + \cdots + b(j,N)z^{-N}}{1 + a(j,1)z^{-1} + \cdots + a(j,N)z^{-N}}$$

这里 $a(j,i)$ 和 $b(j,i)$ $(j=1,2,\cdots,L;\ i=0,1,\cdots,N)$ 分别是数字滤波器分母多项式和分子多项式的系数。

当给定数字滤波器系数 $a(j,i)$ 与 $b(j,i)$ 后，即可用上式对输入序列 $x(j)$ 进行数字滤波，从而得到滤波器的输出 $y(k)$

三、使用说明

1. 子函数语句

void filterp(b,a,n,ns,x,y,len,px,py)

2. 形参说明

b ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分子多项式的系数，b[j][i]表示第 j 个 n 阶节的分子多项式的第 i 个系数。

a ——双精度实型二维数组，体积为 ns×(n+1)。存放滤波器分母多项式的系数，a[j][i]表示第 j 个 n 阶节的分母多项式的第 i 个系数。

n ——整型变量。滤波器每节的阶数。

ns ——整型变量。滤波器的 n 阶节数 L。

x ——双精度实型一维数组，长度为 len。存放滤波器的输入序列；在分块处理时，它用子表示当前块内的滤波器的输入序列。

y ——双精度实型一维数组，长度为 len。存放滤波器的输出序列；在分块处理时，它用于表示当前块内的滤波器的输出序列。

len——整型变量。输入序列与输出序列的长度；在分块处理时，它用子表示块的长度。

px ——双精度实型二维数组，体积为 ns×(n+1)。在分块处理时，它用于保存前一

块滤波时的(n+1)个输入序列值。

py ——双精度实型二维数组,体积为 ns×(n-1)。在分块处理时,它用于保存前一块滤波时的 n 个输出序列值。

当输入序列 x(k)很长时,由于计算机内存的限制,常将其分成彼此衔接的若干块进行处理。数组 px 与 py 就是专为分块处理而设置的。px 用于保存前一块滤波时的(n+1)个输入序列值,即 px[j][ ]={x(k), x(k-1),…,x(k-n)};py 用于保存前一块滤波时的 n 个输出序列值,即 py[j][ ]={y(k-1),y(k-2),…,y(k-n)}。通常,我们假定滤波器的初始条件为零,因此数组 px[ ][ ]与 py[ ][ ]在滤波前都要初始化为零。有关分块处理的具体方法见后面的例题。

四、子函数程序(文件名: filterp.c)

```
#include "math.h"
void filterp(b,a,n,ns,x,y,len,px,py)
int n,ns,len;
double a[],b[],x[],y[],px[],py[];
{ int i,j,k,n1;
  double sum;
  n1 = n + 1;
  for (k=0;k<len;k++)
    { y[k] = 0.0; }
  for (j=0;j<ns;j++)
    { for (k=0;k<len;k++)
        { px[j*n1+0] = x[k];
          sum = b[j*n1+0] * px[j*n1+0];
          for (i=1;i<=n;i++)
          { sum += b[j*n1+i]*px[j*n1+i]-a[j*n1+i]*py[j*n1+i];}
          if ( fabs(sum) > 1.0e10 )
            { printf("This is an unstable filter ! \n");
              exit(0);
            }
          for (i=n;i>=2;i--)
            { px[j*n1+i] = px[j*n1+i-1];
              py[j*n1+i] = py[j*n1-i-1];
            }
          px[j*n1+1] = px[j*n1+0];
          py[j*n1+1] = sum;
          y[k] = y[k] + sum;
        }
```

```
    }
}
```

五、例　题

4 阶切比雪夫低通数字滤波器的传递函数为

$$H(z)=\frac{0.08327+0.0239z^{-1}}{1-1.5658z^{-1}+0.6549z^{-2}}-\frac{0.08327+0.0246z^{-1}}{1-1.4934z^{-1}+0.8392z^{-2}}$$

它由两个 2 阶节并联而成。选取参数 n=2，ns=2，共分 4 块进行处理，每块长度 len=25。求该滤波器的单位冲激响应，并画出相应的图形。

主函数程序(文件名:filterp.m)：

```
#include "stdio.h"
#include "filterp.c"
main()
{ int i,j,n,ns,len,nblk;
  double b[2][3]={{0.08327, 0.0239, 0.0}, {-0.08327, -0.0246, 0.0}};
  double a[2][3]={{1.0, -1.5658, 0.6549}, {1.0, -1.4934, 0.8392}};
  double px[2][3],py[2][3];
  double x[25],y[25],data[100];
  FILE *fp;
  n = 2;
  ns = 2;
  len = 25;
  nblk = 4;
  x[0] = 1.0;
  for (i=1;i<len;i++ )
    { x[i]=0.0; }
  for (i=0;i<ns;i++)
  for (j=0;j<=n;j++)
    { px[i][j]=0.0;
      py[i][j]=0.0;
    }
  for (j=0;j<nblk;j++)
    { filterp(b,a,n,ns,x,y,len,px,py);
      for (i=0;i<len;i++)
        { data[j*len+i] = y[i];
          x[i] = 0.0;
        }
    }
```

```
    printf("Unit Impulse Response\n");
    for (i=0;i<16;i++)
      { printf("         %10.7lf",data[i]);
        if ( i%4==3 ) printf("\n");
      }
    if ( ( fp = fopen("filterp.dat","w")) == NULL )
      { printf("cannot open file 'filterp.dat' ! \n");
        exit(0);
      }
    for (i=0;i<100;i++)
      { fprintf(fp,"%2d        %lf\n",i,data[i]); }
    fclose(fp);
  }
```

运行结果：

单位冲激响应(前 16 点)为

0.0000000	0.0053287	0.0344748	0.0889894
0.1523266	0.2010379	0.2162496	0.1913117
0.1335624	0.0605640	−0.0069497	−0.0523818
−0.0682015	−0.0571050	−0.0295577	0.0008975

单位冲激响应如图 2—4—7 所示。

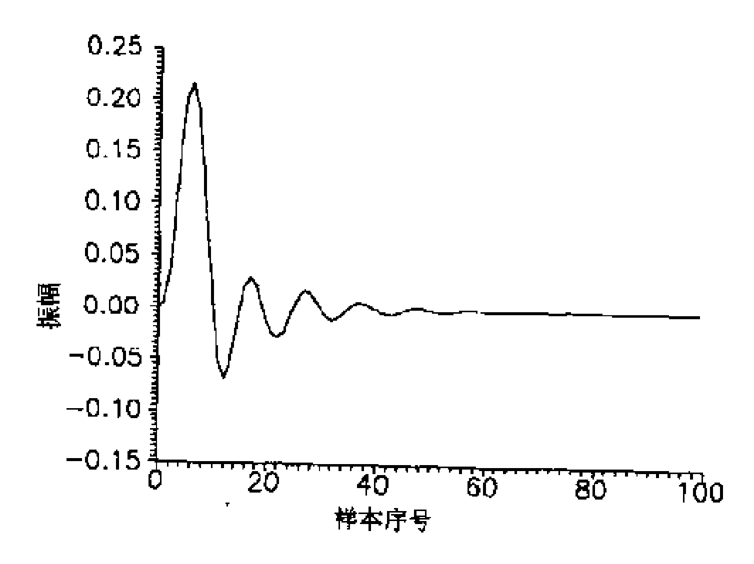


图 2—4—7 并联型数字滤波器的单位冲激响应

第五章 IIR 数字滤波器的设计

§5.1 巴特沃兹和切比雪夫数字滤波器的设计

一、功 能

根据模拟滤波器理论，用双线性变换法设计低通、高通、带通和带阻的巴特沃兹、切比雪夫和逆切比雪夫 IIR 数字滤波器。

二、方法简介

设计巴特沃兹和切比雪夫数字滤波器的步骤如下：

1. 归一化模拟低通滤波器的设计

(1). 巴特沃兹滤波器

归一化 L(L 为偶数)阶巴特沃兹模拟低通滤波器的传递函数为

$$H(s)=\prod_{k=1}^{L/2}\frac{1}{s^2-2s\cos\left(\frac{2k+L-1}{2L}\pi\right)+1}$$

(2). 切比雪夫滤波器

归一化 L(L 为偶数)阶切比雪夫模拟低通滤波器的传递函数为

$$H(s)=A\frac{s_1s_2\cdots s_L}{(s-s_1)(s-s_2)\cdots(s-s_L)}$$

其中系数 A 为

$$A=\frac{1}{\sqrt{1+\varepsilon^2}}$$

$H(s)$的极点 s_k 为

$$s_k=\sigma_k+j\,\Omega_k,\quad k=1,2,\cdots,L$$

式中

$$\sigma_k=\frac{1}{2}(\gamma^{-1}-\gamma)\sin\left(\frac{2k-1}{2L}\pi\right)$$

$$\Omega_k=\frac{1}{2}(\gamma^{-1}+\gamma)\cos\left(\frac{2k-1}{2L}\pi\right)$$

$$\gamma=\left(\frac{1+\sqrt{1+\varepsilon^2}}{\varepsilon}\right)^{1/L}$$

(3). 逆切比雪夫滤波器

归一化 L(L 为偶数)阶逆切比雪夫模拟低通滤波器的传递函数为 $WTBZ$]

$$H(s)=\frac{s_1s_2\cdots s_L(s-t_1)(s-t_2)\cdots(s-t_L)}{t_1t_2\cdots t_L(s-s_1)(s-s_2)\cdots(s-s_L)}$$

其中 $H(s)$的零点 t_k 和极点 s_k 为

$$t_k=j\frac{\Omega_s}{\cos[\pi(2k-1)/(2L)]}$$

$$s_k=\sigma_k+j\Omega_k,\quad k=1,2,\cdots,L$$

$$\sigma_k=\frac{\alpha_k\Omega_s}{\alpha_k^2+\beta_k^2}$$

$$\Omega_k=\frac{-\beta_k\Omega_s}{\alpha_k^2+\beta_k^2}$$

$$\alpha_k=\frac{1}{2}(\gamma^{-1}-\gamma)\sin\left(\frac{2k-1}{2L}\pi\right)$$

$$\beta_k=\frac{1}{2}(r^{-1}+\gamma)\cos\left(\frac{2k-1}{2L}\pi\right)$$

$$\gamma=\left(\lambda+\sqrt{\lambda^2-1}\right)^{1/L}$$

在上面诸式中，Ω_s 是阻带的边界频率，λ 是与阻带衰减相关的参数。在 $\Omega=\Omega_s$ 处，滤波器的幅度增益为 $1/\lambda$。

显而易见，把共轭极点进行组合后，巴特沃兹、切比雪夫和逆切比雪夫滤波器都可写成$\frac{L}{2}$个 2 阶节级联的形式。

2. 模拟滤波器的频率变换

低通到高通的变换：

$$s\longleftarrow\frac{1}{s}$$

低通到带通的变换：

$$s\longleftarrow\frac{s^2+\Omega_0^2}{sW}$$

低通到带阻的变换：

$$\begin{cases}s\longleftarrow\dfrac{1}{s}\\[2ex] s\longleftarrow\dfrac{s^2+\Omega_0^2}{sW}\end{cases}$$

其中

$$W=\Omega_H-\Omega_L$$

$$\Omega_0=\sqrt{\Omega_H\Omega_L}$$

这里 Ω_L 和 Ω_H 分别是带通或带阻滤波器的通带下边界频率和通带上边界频率。

3. 模拟滤波器到数字滤波器的变换

(1) 计算预畸变后的模拟滤波器的通带边界频率

$$\Omega_c=\tan\left(\frac{\omega_c}{2}\right)=\tan(\pi f_c/f_s)$$

(2) 模拟滤波器的频率轴尺度变换：

$$\hat{H}(s) = H\left(\frac{s}{\Omega_c}\right)$$

(3)用双线性变换法，将模拟滤波器 $\hat{H}(s)$变换为数字滤波器 $H(z)$

$$H(z) = \hat{H}(s)|_{s=(1-z^{-1})/(1+z^{-1})}$$

三、使用说明

1. 子函数语句

void iirbcf(ifilt,band,ns,n,f1,f2,f3,f4,db,b,a)

2. 形参说明

ifilt ——整型变量。滤波器的类型。取值为1、2和3，分别对应切比雪夫、逆切比雪夫和巴特沃兹滤波器。

band ——整型变量。滤波器的通带形式。取值为1、2、3和4，分别对应低通、高通、带通和带阻滤波器。

ns ——整型变量。滤波器的n阶节数。

n ——整型变量。滤波器每节的阶数。对于低通和高通滤波器，n=2；对于带通和带阻滤波器，n=4。

f1 ——双精度实型变量。

f2 ——双精度实型变量。

f3 ——双精度实型变量。

f4 ——双精度实型变量。

对于巴特沃兹滤波器：低通时，f1是通带边界频率，f2=f3=f4=0；高通时，f2是通带边界频率，f1=f3=f4=0；带通时，f2是通带下边界频率，f3是通带上边界频率，f1=f4=0；带阻时，f1是通带下边界频率，f4是通带上边界频率，f2=f3=0。

对于切比雪夫滤波器：低通时，f1是通带边界频率，f2是阻带边界频率，f3=f4=0；高通时，f2是通带边界频率，f1是阻带边界频率，f3=f4=0；带通时，f2是通带下边界频率，f3是通带上边界频率，f1是阻带下边界频率，f4是阻带上边界频率；带阻时，f1是通带下边界频率，f4是通带上边界频率，f2是阻带下边界频率，f3是阻带上边界频率。

db ——双精度实型变量。滤波器的阻带衰减(用dB表示)。

b ——双精度实型二维数组，体积为ns×(n+1)。存放滤波器分子多项式的系数。b[j][i]表示第j个n阶节的分子多项式的第i个系数。

a ——双精度实型二维数组，体积为ns×(n+1)。存放滤波器分母多项式的系数。a[j][i]表示第i个n阶节的分母多项式的第i个系数。

四、子函数程序(文件名:iirbcf.c)

```
#include "math.h"
void iirbcf(ifilt,band,ns,n,f1,f2,f3,f4,db,b,a)
double b[],a[],f1,f2,f3,f4,db;
int ifilt,band,ns,n;
```

```
{ int k;
  double omega,lamda,epslon,fl,fh;
  double d[5],c[5];
  void chebyi(),chebyii(),bwtf();
  double cosh1(),warp(),bpsub(),omin();
  void fblt();
  if ( (band == 1) || (band==4) ) fl = f1;
  if ( (band == 2) || (band==3) ) fl = f2;
  if (band <= 3) fh = f3;
  if (band == 4) fh = f4;
  if ( ifilt < 3 )
    { switch ( band )
        { case 1:
          case 2:
            { omega=warp(f2)/warp(f1);
              break;
            }
          case 3:
            { omega=omin(bpsub(warp(f1),fh,fl),bpsub(warp(f4),fh,fl));
              break;
            }
          case 4:
    { omega=omin(1.0/bpsub(warp(f2),fh,fl),1.0/bpsub(warp(f3),fh,fl));}
        }
      lamda = pow(10.0,(db/20.0));
      epslon = lamda/cosh(2 * ns * cosh1(omega));
    }
   for (k=0;k<ns;k++)
    { switch ( ifilt )
        { case 1:
            { chebyi(2 * ns,k,4,epslon,d,c);
              break;
            }
          case 2:
            { chebyii(2 * ns,k,4,omega,lamda,d,c);
              break;
            }
          case 3:
```

```
            { bwtf(2 * ns,k,4,d,c);
              break;
            }
        }
      fblt(d,c,n,band,fl,fh,&b[k * (n+1)+0],&a[k * (n+1)+0]);
    }
}

static double cosh1(x)
double x;
{ double z;
  z = log(x+sqrt(x * x-1.0));
  return(z);
}

static double warp(f)
double f;
{ double pi,z;
  pi = 4.0 * atan(1.0);
  z = tan(pi * f);
  return(z);
}

static double bpsub(om,fh,fl)
double om,fh,fl;
{ double z;
  z = (om * om-warp(fh) * warp(fl))/((warp(fh)-warp(fl)) * om);
  return(z);
}

static double omin(om1,om2)
double om1,om2;
{ double z,z1,z2;
  z1 = fabs(om1);
  z2 = fabs(om2);
  z = ( z1 < z2 ) ? z1 : z2;
  return(z);
}
```

```
static void bwtf(ln,k,n,d,c)
int ln,k,n;
double d[],c[];
{ int i;
  double pi,tmp;
  pi = 4.0 * atan(1.0);
  d[0] = 1.0;
  c[0] = 1.0;
  for(i=1;i<=n;i++)
    { d[i] = 0.0;
      c[i] = 0.0;
    }
  tmp = (k+1)-(ln+1.0)/2.0;
  if (tmp == 0.0)
    { c[1] = 1.0; }
  else
    { c[1] = -2.0 * cos((2 * (k+1)+ln-1) * pi/(2 * ln));
      c[2] = 1.0;
    }
}

static void chebyi(ln,k,n,ep,d,c)
double d[],c[],ep;
int ln,k,n;
{ int i;
  double pi,gam,omega,sigma;
  pi = 4.0 * atan(1.0);
  gam = pow(((1.0 + sqrt(1.0+ep * ep))/ep), 1.0/ln);
  sigma = 0.5 * (1.0/gam-gam) * sin((2 * (k+1)-1) * pi/(2 * ln));
  omega = 0.5 * (1.0/gam+gam) * cos((2 * (k+1)-1) * pi/(2 * ln));
  for (i=0;i<=n;i++)
    { d[i] = 0.0;
      c[i] = 0.0;
    }
  if ( ((ln%2)==1) && ((k+1) == (ln+1)/2) )
    { d[0] = -sigma;
      c[0] = d[0];
```

```
        c[1] = 1.0;
      }
    else
      { c[0] = sigma * sigma + omega * omega;
        c[1] = -2.0 * sigma;
        c[2] = 1.0;
        d[0] = c[0];
        if ( ((ln%2)==0) && (k == 0) )
          d[0] = d[0]/sqrt(1.0+ep * ep);
      }
}

static void chebyii(ln,k,n,ws,att,d,c)
double d[],c[],ws,att;
int ln,k,n;
{ int i;
  double pi,gam,alpha,beta,sigma,omega,scln,scld;
  pi = 4.0 * atan(1.0);
  gam = pow((att+sqrt(att * att-1.0)),1.0/ln);
  alpha = 0.5 * (1.0/gam-gam) * sin((2 * (k+1) -1) * pi/(2 * ln));
  beta = 0.5 * (1.0/gam+gam) * cos((2 * (k+1)-1) * pi/(2 * ln));
  sigma = ws * alpha/(alpha * alpha + beta * beta);
  omega = -1.0 * ws * beta/(alpha * alpha + beta * beta);
  for (i=0;i<=n;i++)
    { d[i] = 0.0;
      c[i] = 0.0;
    }
  if ( ((ln%2)==1) && ((k+1)==(ln+1)/2) )
    { d[0] = -1.0 * sigma;
      c[0] = d[0];
      c[1] = 1.0;
    }
  else
    { scln = sigma * sigma + omega * omega;
      scld = pow((ws/cos((2 * (k+1)-1) * pi/(2 * ln))),2);
      d[0] = scln * scld;
      d[2] = scln;
      c[0] = d[0];
```

```
        c[1] = -2.0 * sigma * scld;
        c[2] = scld;
      }
}

#include "stdlib.h"
static void fblt(d,c,n,band,fln,fhn,b,a)
int n,band;
double fln,fhn,d[],c[],b[],a[];
{ int i,k,m,n1,n2,ls;
  double pi,w,w0,w1,w2,tmp,tmpd,tmpc, * work;
  double combin();
  void bilinear();
  pi = 4.0 * atan(1.0);
  w1 = tan(pi * fln);
  for (i=n;i>=0;i--)
    { if ( (c[i] != 0.0) || (d[i] != 0.0) )
        break;
    }
  m = i;
  switch ( band )
    { case 1:
      case 2:
        { n2 = m;
          n1 = n2 + 1;
          if (band == 2)
            { for (i=0;i<=m/2;i++)
                { tmp = d[i];
                  d[i] = d[m-i];
                  d[m-i] = tmp;
                  tmp = c[i];
                  c[i] = c[m-i];
                  c[m-i] = tmp;
                }
            }
          for (i=0;i<=m;i++)
            { d[i] = d[i]/pow(w1,i);
              c[i] = c[i]/pow(w1,i);
```

```
        }
        break;
      }
    case 3:
    case 4:
      { n2 = 2 * m;
        n1 = n2 + 1;
        work = malloc(n1 * n1 * sizeof(double));
        w2 = tan(pi * fhn);
        w = w2 - w1;
        w0 = w1 * w2;
        if (band == 4)
          { for (i=0;i<=m/2;i++)
              { tmp = d[i];
                d[i] = d[m-i];
                d[m-i] = tmp;
                tmp = c[i];
                c[i] = c[m-i];
                c[m-i] = tmp;
              }
          }
        for (i=0;i<=n2;i++)
          { work[0 * n1+i] = 0.0;
            work[1 * n1+i] = 0.0;
          }
        for (i=0;i<=m;i++)
          { tmpd = d[i] * pow(w,(m-i));
            tmpc = c[i] * pow(w,(m-i));
            for (k=0;k<=i;k++)
              { ls = m+i-2 * k;
                tmp = combin(i,i)/(combin(k,k) * combin(i-k,i-k));
                work[0 * n1+ls] += tmpd * pow(w0,k) * tmp;
                work[1 * n1+ls] += tmpc * pow(w0,k) * tmp;
              }
          }
        for (i=0;i<=n2;i++)
          { d[i] = work[0 * n1+i];
            c[i] = work[1 * n1+i];
```

```
            }
          free(work);
        }
    }
  bilinear(d,c,b,a,n);
}

static double combin(i1,i2)
int i1,i2;
{ int i;
  double s;
  s=1.0;
  if ( i2 == 0 ) return (s);
  for (i=i1;i>(i1-i2);i--)
    { s *= i; }
  return (s);
}

static void bilinear(d,c,b,a,n)
int n;
double d[],c[],b[],a[];
{ int i,j,n1;
  double sum,atmp,scale, *temp;
  n1 = n + 1;
  temp = malloc(n1 * n1 * sizeof(double));
  for (j=0;j<=n;j++)
    { temp[j*n1+0] = 1.0; }
  sum = 1.0;
  for (i=1;i<=n;i++)
    { sum = sum * (double)(n-i+1)/(double)i;
      temp[0*n1+i] = sum;
    }
  for (i=1;i<=n;i++)
  for (j=1;j<=n;j++)
    { temp[j*n1+i] = temp[(j-1)*n1+i]-temp[j*n1+i-1]
                        -temp[(j-1)*n1+i-1]; }
  for (i=n;i>=0;i--)
    { b[i]=0.0;
```

```
        atmp=0.0;
        for (j=0;j<=n;j++)
          { b[i]=b[i] + temp[j*n1+i]*d[j];
            atmp=atmp + temp[j*n1+i]*c[j];
          }
        scale = atmp;
        if (i != 0) a[i]=atmp;
      }
    for (i=0;i<=n;i++)
      { b[i] = b[i]/scale;
        a[i] = a[i]/scale;
      }
    a[0] = 1.0;
    free(temp);
}
```

五、例　题

下面给出主函数程序，它调用 iirbcf.c 子函数。通过人机对话方式输入参数后，它可以设计巴特沃兹、切比雪夫和逆切比雪夫滤波器，每种滤波器都具有低通、高通、带通、带阻这四种形式。下面对输入参数进行说明：

ifilt：滤波器的类型；band：滤波器的通带；ns：滤波器的 n 阶节数；fs：采样频率；db：阻带衰减；fname：幅频响应的文件名。

对于低通和高通滤波器　fc：通带边界频率；fr：阻带边界频率。

对于带通和带阻滤波器　flc：通带下边界频率；fhc：通带上边界频率；fls：阻带下边界频率；fhs：阻带上边界频率。

主函数要调用计算级联型数字滤波器频率响应的函数 gainc()，参看第二篇 §4.2 节。

主函数程序（文件名：iirbcf.m）：

```
#include "stdio.h"
#include "iirbcf.c"
#include "gainc.c"
main()
{ int i,k,n,ns,ifilt,band;
  double a[50],b[50],x[300],y[300];
  double f1,f2,f3,f4,fc,fr,fs,flc,fls,fhc,fhs,freq,db;
  char fname[40];
  FILE *fp;
  printf("enter 1 for Cbebyshev I, 2 for Chebyshev II, 3 for Butterworth\n");
  scanf("%d",&ifilt);
```

```
printf("enter 1 for lowpass, 2 for highpass, 3 for bandpass, 4 for bandstop\n");
scanf("%d",&band);
n = ( band <= 2 ) ? 2 : 4;
printf("enter the number of filter section\n");
scanf("%d",&ns);
printf("enter sample frequence fs\n");
scanf("%lf",&fs);
if ( ifilt <= 2 )
  { switch ( band )
      { case 1:
        case 2:
          { printf("enter passband edge frequency fc\n");
            scanf("%lf",&fc);
            printf("enter stopband edge frequency fr\n");
            scanf("%lf",&fr);
            if ( band == 1 )
              { f1 = fc;
                f2 = fr;
              }
            else
              { f1 = fr;
                f2 = fc;
              }
            f3 = f4 = 0.0;
            break;
          }
        case 3:
        case 4:
          { printf("enter the lower passband edge frequency flc\n");
            scanf("%lf",&flc);
            printf("enter the higher passband edge frequency fhc\n");
            scanf("%lf",&fhc);
            printf("enter the lower stopband edge frequency fls\n");
            scanf("%lf",&fls);
            printf("enter the higher stopband edge frequency fhs\n");
            scanf("%lf",&fhs);
            if ( band == 3 )
              { f1 = fls;
```

```
              f2 = flc;
              f3 = fhc;
              f4 = fhs;
            }
          else
            { f1 = flc;
              f2 = fls;
              f3 = fhs;
              f4 = fhc;
            }
        }
      }
    printf("enter stopband attenuation (dB)\n");
    scanf("%lf",&db);
  }
else
  { switch ( band )
      { case 1:
        case 2:
          { printf("enter passband edge frequency fc\n");
            scanf("%lf",&fc);
            f1 = f2 = f3 = f4 = 0.0;
            if ( band == 1 )
              { f1 = fc; }
            else
              { f2 = fc; }
            break;
          }
        case 3:
        case 4:
          { printf("enter the lower passband edge frequency flc\n");
            scanf("%lf",&flc);
            printf("enter the higher passband edge frequency fhc\n");
            scanf("%lf",&fhc);
            f1 = f2 = f3 = f4 = 0.0;
            if ( band == 3 )
              { f2 = flc;
                f3 = fhc;
```

```
                }
              else
                { f1 = flc;
                  f4 = fhc;
                }
            }
        }
    }
  f1 = f1/fs;
  f2 = f2/fs;
  f3 = f3/fs;
  f4 = f4/fs;
  iirbcf(ifilt,band,ns,n,f1,f2,f3,f4,db,b,a);
  for (k=0;k<ns;k++)
    { printf("\nsection %d\n\n",k+1);
      for (i=0;i<=n;i++)
        { printf("     b[%d][%d] = %10.7lf ",k,i,b[k*(n+1)+i]);
          if ( ((i%2)==0) && (i != 0) ) printf("\n");
        }
      printf("\n");
      for (i=0;i<=n;i++)
        { printf("     a[%d][%d] = %10.7lf ",k,i,a[k*(n+1)+i]);
          if ( ((i%2) == 0) && (i != 0) ) printf("\n");
        }
      printf("\n");
    }
  printf("\nenter file name of magnitude response\n");
  scanf(" %s",fname);
  if ((fp=fopen(fname,"w"))==NULL)
    { printf("cannot open file %s\n",fname);
      exit(0);
    }
  gainc(b,a,n,ns,x,y,300,2);
  for (i=0;i<300;i++)
    { freq = i*0.5/300.0;
      fprintf(fp," %lf      %lf\n",freq,x[i]);
    }
  fclose(fp);
```

}

运行结果：

例 1：巴特沃兹低通数字滤波器的设计。选择参数 ifilt=3，band=1，ns=4，fc=0.2，fr=0.3，fs=1。运行程序得到滤波器的系数为

第一级：

b[0][0] = 0.2914207　　b[0][1] = 0.5828415　　b[0][2] = 0.2914207

a[0][0] = 1.0000000　　a[0][1] =－0.5213093　　a[0][2] = 0.6869922

第二级

b[1][0] = 0.2260510　　b[1][1] = 0.4521020　　b[1][2] = 0.2260510

a[1][0] = 1.0000000　　a[1][1] =－0.4043723　　a[1][2] = 0.3085762

第三级

b[2][0] = 0.1929285　　b[2][1] = 0.3858571　　b[2][2] = 0.1929285

a[2][0] = 1.0000000　　a[2][1] =－0.3451210　　a[2][2] = 0.1168351

第四级

b[3][0] = 0.1787535　　b[3][1] = 0.3575069　　b[3][2] = 0.1787535

a[3][0] = 1.0000000　　a[3][1] =－0.3197639　　a[3][2] = 0.0347777

该滤波器的幅频响应如图 2—5—1 所示。

例 2：巴特沃兹高通数字滤波器的设计。选择参数 ifilt=3，band=2，ns=4，fc=0.3，fr=0.2，fs=1。该滤波器的幅频响应如图 2—5—2 所示。

例 3：巴特沃兹带通数字滤波器的设计。选择参数 ifilt=3，band=3，ns=4，flc=0.1，fhc=0.2，fs=1。该滤波器的幅频响应如图 2—5—3 所示。

例 4：巴特沃兹带阻数字滤波器的设计。选择参数 ifilt=3，band=4，ns=4，flc=0.1，fhc=0.3，fs=1。该滤波器的幅频响应如图 2—5—4 所示。

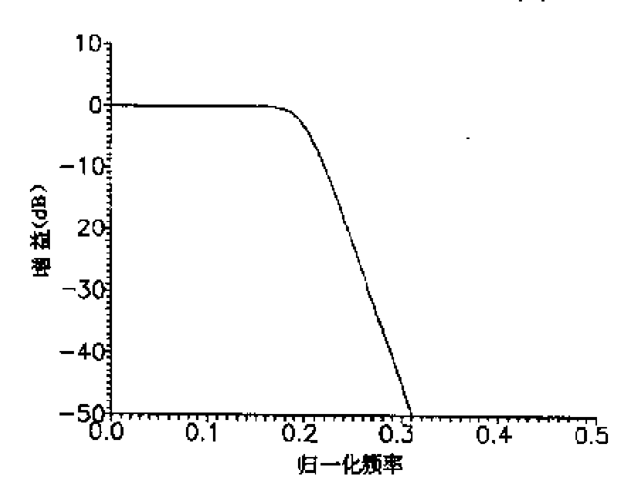


图 2—5—1　巴特沃兹低通数字滤波器的幅频响应

例 5：切比雪夫低通数字滤波器的设计。选择参数 ifilt=1，band=1，ns=2，fc=0.2，fr=0.3，fs=1，db=40。该滤波器的幅频响应如图 2—5—5 所示。

例 6：切比雪夫高通数字滤波器的设计。选择参数 ifilt=1，band=2，ns=2，fc=

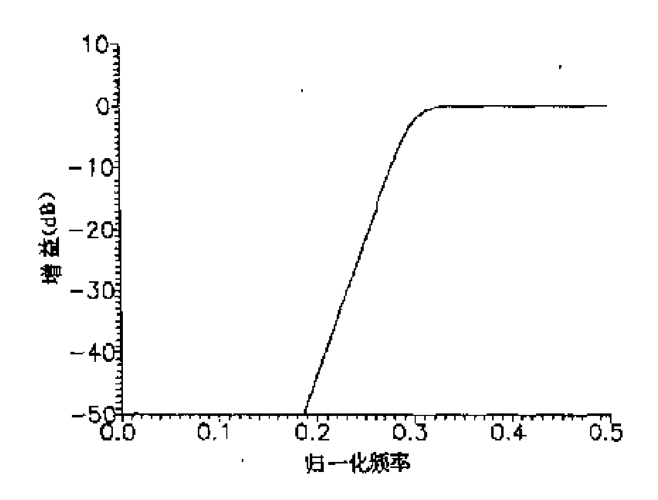


图 2－5－2　巴特沃兹高通数字滤波器的幅频响应

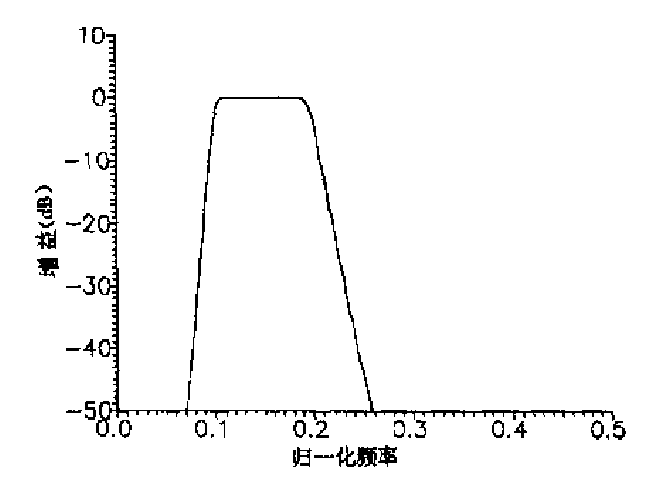


图 2－5－3　巴特沃兹带通数字滤波器的幅频响应

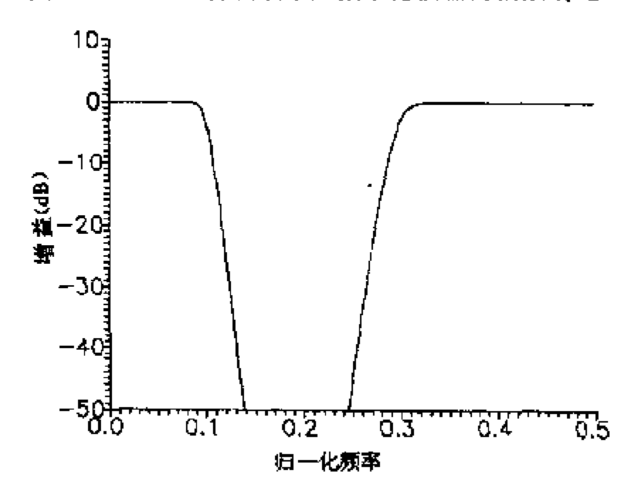


图 2－5－4　巴特沃兹带阻数字滤波器的幅频响应

0.3，fr＝0.2，fs＝1，db＝40。运行程序得到滤波器的系数为

第一级：

b[0][0] = 0.1795492　　b[0][1] = −0.3590984　　b[0][2] = 0.1795492

a[0][0] = 1.0000000　　a[0][1] = 0.6820193　　a[0][2] = 0.8756792

第二级：

b[1][0] = 0.0703521　　b[1][1] = −0.1407042　　b[1][2] = 0.0703521

a[1][0] = 1.0000000　　a[1][1] = 1.3651728　　a[1][2] = 0.6465812

该滤波器的幅频响应如图 2—5—6 所示。

例 7：切比雪夫带通数字滤波器的设计。选择参数 ifilt=1，band=3，ns=2，flc=0.1，fhc=0.2，fls=0.05，fhs=0.3，fs=1，db=40。该滤波器的幅频响应如图 2—5—7 所示。

例 8：切比雪夫带阻数字滤波器的设计。选择参数 ifilt=1，band=4，ns=2，flc=0.1，fhc=0.3，fls=0.2，fhs=0.25，fs=1，db=40。该滤波器的幅频响应如图 2—5—8 所示。

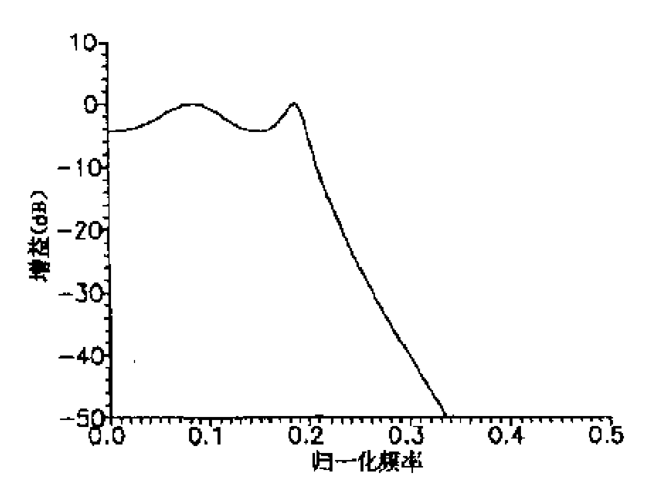


图 2—5—5　切比雪夫低通数字滤波器的幅频响应

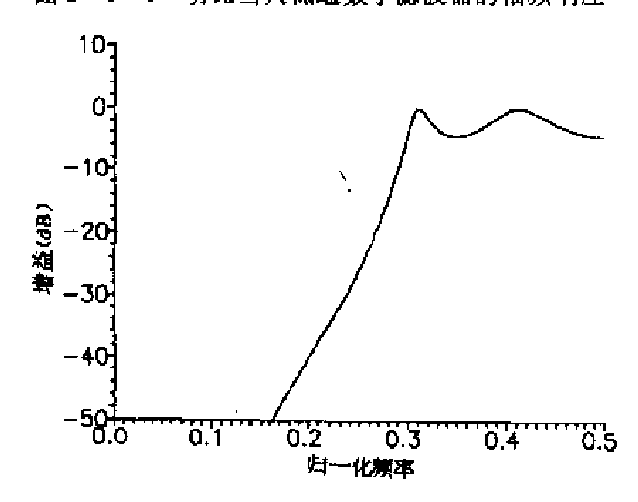


图 2—5—6　切比雪夫高通数字滤波器的幅频响应

例 9：逆切比雪夫低通数字滤波器的设计。选择参数 ifilt=2，band=1，ns=2，fc=0.2，fr=0.3，fs=1，db=40。该滤波器的幅频响应如图 2—5—9 所示。

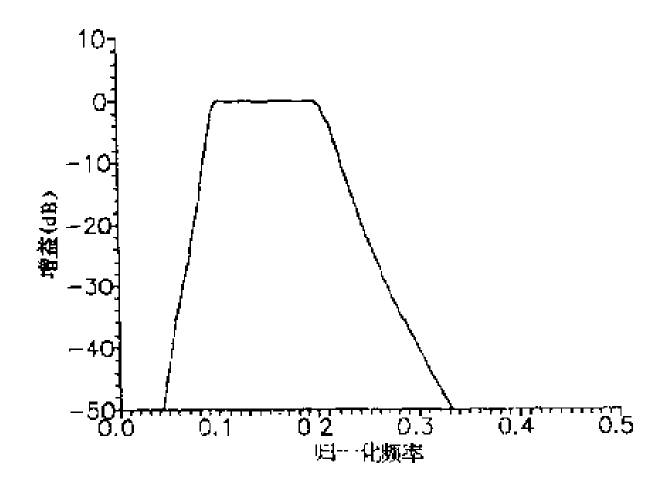


图 2—5—7　切比雪夫带通数字滤波器的幅频响应

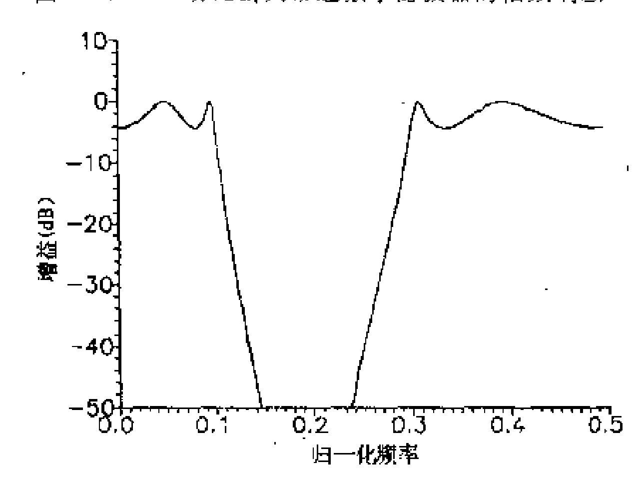


图 2—5—8　切比雪夫带阻数字滤波器的幅频响应

例 10：逆切比雪夫高通数字滤波器的设计。选择参数 ifilt＝2，band＝2，ns＝2，fc＝0.3，fr＝0.2，fs＝1，db＝40。该滤波器的幅频响应如图 2—5—10 所示。

例 11：逆切比雪夫带通数字滤波器的设计。选择参数 ifilt＝2，band＝3，ns＝2，flc＝0.1，fhc＝0.2，fls＝0.05，fhs＝0.3，fs＝1，db＝40。运行程序得到滤波器的系数为

第一级：

b[0][0] ＝　0.2927875　　b[0][1] ＝－0.3425226　　b[0][2] ＝　0.1803287

b[0][3] ＝－0.3425226　　b[0][4] ＝　0.2927875

a[0][0] ＝　1.0000000　　a[0][1] ＝－1.8292883　　a[0][2] ＝　1.9287603

a[0][3] ＝－1.3061690　　a[0][4] ＝　0.5767876

第二级：

b[1][0] ＝　0.1502828　　b[1][1] ＝－0.0496149　　b[1][2] ＝－0.1896233

b[1][3] ＝－0.0496149　　b[1][4] ＝　0.1502828

a[1][0] ＝　1.0000000　　a[1][1] ＝－1.6171168　　a[1][2] ＝　1.2865002

a[1][3] ＝－0.5508877　　a[1][4] ＝　0.1374025

该滤波器的幅频响应如图 2－5－11 所示。

例 12：逆切比雪夫带阻数字滤波器的设计。选择参数 ifilt＝2，band＝4，ns＝2，flc＝0.1，fhc＝0.3，fls＝0.2，fhs＝0.25，fs－1，db＝40。该滤波器的幅频响应如图 2－5－12 所示。

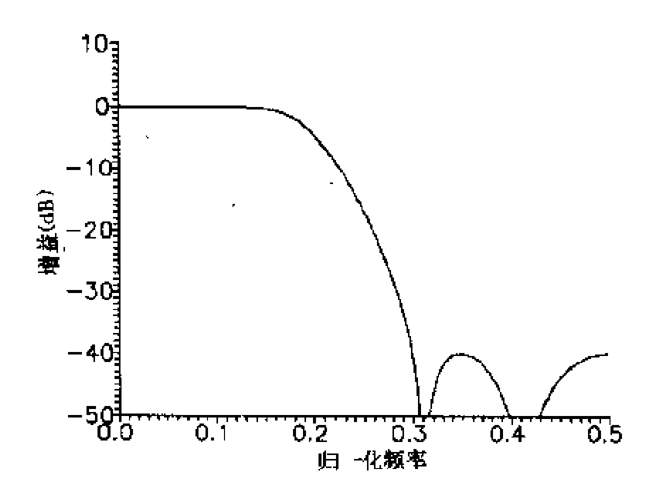


图 2－5－9　逆切比雪夫低通数字滤波器的幅频响应

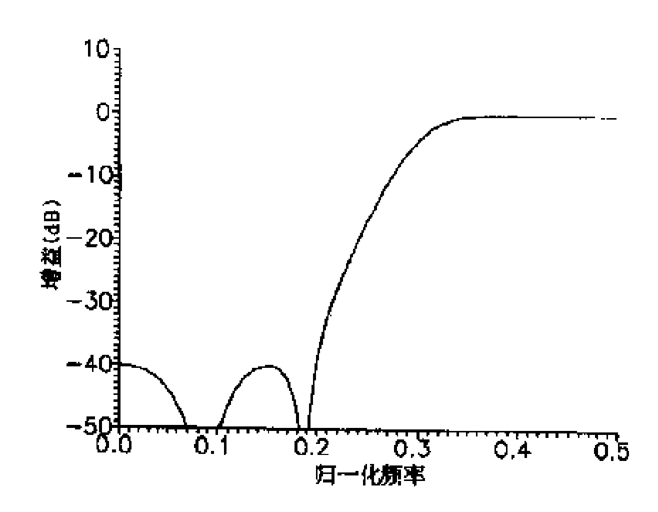


图 2－5－10　逆切比雪夫高通数字滤波器的幅频响应

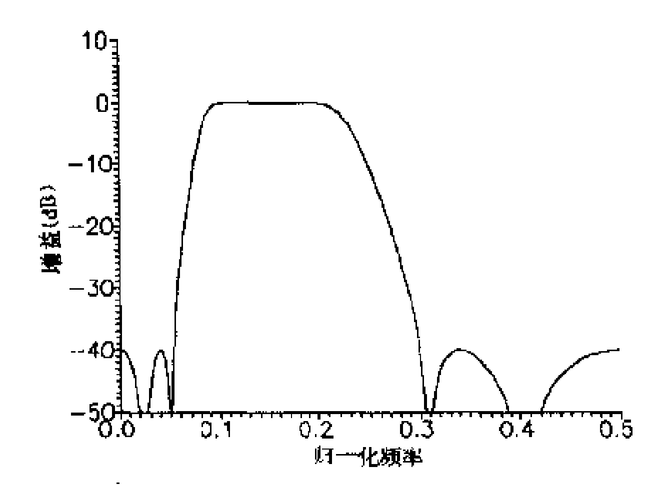


图 2—5—11　逆切比雪夫带通数字滤波器的幅频响应

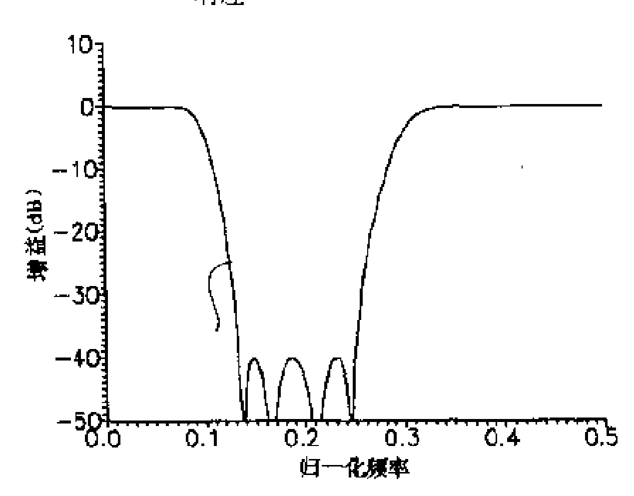


图 2—5—12　逆切比雪夫带阻数字滤波器的幅频响应

§5.2　任意幅度 IIR 数字滤波器的优化设计

一、功　能

用 K. Steiglitz 方法，优化设计任意幅度 IIR 数字滤波器。

二、方法简介

级联型 IIR 数字滤波器的传递函数为

$$H(z) = c\prod_{k=1}^{L}\frac{1 + a_k z^{-1} + b_k z^{-2}}{1 + c_k z^{-1} + d_k z^{-2}}$$

式中 c 是增益因子，a_k，b_k，c_k 和 d_k 是数字滤波器的系数。

假设要求的幅频响应为 $H_d(z_i)$，那么滤波器设计的准则就是在一组离散频率点

$$z_i = e^{j2\pi f_i}, \qquad i=1,2,\cdots,M$$

上，使频域均方误差 E 最小：

$$E = \sum_{i=1}^{M}[|H(z_i)| - H_d(z_i)]^2$$

用 x 表示 $4L$ 个未知常数 $\{a_k,b_k,c_k,d_k\}$，$k=1,2,\cdots,L$，即

$$x = (a_1,b_1,c_1,d_1,\cdots,a_L,b_L,c_L,d_L)^T$$

于是传递函数可写为

$$H(z,c,x) = c\prod_{k=1}^{L}\frac{1+a_k z^{-1}+b_k z^{-2}}{1+c_k z^{-1}+d_k z^{-2}} = c\,H(z,x)$$

均方误差函数为

$$E(c,\mathbf{x}) = \sum_{i=1}^{M}[\,c\,|H(z_i,\mathbf{x})| - H_d(z_i)\,]^2$$

假设使 $E(c,x)$ 最小的变量值为 x^* 和 c^*，那么最优增益因子 c^* 为

$$c^* = \frac{\sum_{i=1}^{M}|H(z_i,x)|\;H_d(z_i)}{\sum_{i=1}^{M}|H(z_i,x)|^2}$$

求 x^* 必须运用非线性优化方法，此处采用 DFP 变尺度算法。为了应用这种方法，需要 $E(c^*,x)$ 对 $4L$ 个 x 分量的每一个求偏导数，即

$$\frac{\partial E(c^*,x)}{\partial x_i} = 2\,c^*\sum_{i=1}^{M}[c^*|H(z_i,x)| - H_d(z_i)]\frac{\partial|H(z_i,x)|}{\partial x_i}$$

式中 x_i 是 x 的第 i 个分量。具体可以用下式表示

$$\frac{\partial|H(z_i,x)|}{\partial a_k} = |H(z_i,x)|\ \mathrm{Re}\left[\frac{z_i^{-1}}{1+a_k z_i^{-1}+b_k z_i^{-2}}\right]$$

$$\frac{\partial|H(z_i,x)|}{\partial b_k} = |H(z_i,x)|\ \mathrm{Re}\left[\frac{z_i^{-2}}{1+a_k z_i^{-1}+b_k z_i^{-2}}\right]$$

$$\frac{\partial|H(z_i,x)|}{\partial c_k} = -|H(z_i,x)|\ \mathrm{Re}\left[\frac{z_i^{-1}}{1+c_k z_i^{-1}+d_k z_i^{-2}}\right]$$

$$\frac{\partial|H(z_i,x)|}{\partial d_k} = -|H(z_i,x)|\ \mathrm{Re}\left[\frac{z_i^{-2}}{1+c_k z_i^{-1}+d_k z_i^{-2}}\right]$$

如果滤波器有的极点位于单位圆外，那么可用一个全通网络把这些极点反演到单位圆内，这样既保证滤波器的稳定性，又不改变其幅度响应。

三、使用说明

1. 子函数语句

void iircad(fname)

2. 形参说明

fname —— 字符型指针变量。数字滤波器技术指标的数据文件名。

函数 iircad()调用以下函数：

子函数 funct()，计算最优增益因子 c^*，均方误差 E 及 DFP 算法中所用到的偏导数，并输出滤波器的各个参数。

子函数 roots()，计算滤波器的零点和极点。

子函数 inside()，将单位圆外的根反演到单位圆内。

子函数 dfmfp()，DFP 优化算法，确定 $E(c^*,x)$的最小值以及最优参数 c^* 和 x^*。

子函数 gainc()，计算滤波器的频率响应。参看第二篇 § 4.2 节。

四、子函数程序(文件名:iircad.c)

```
#include "math.h"
#include "stdio.h"
#include "gainc.c"
int call;
void iircad(fname)
char  *fname;
{ int j,m,m1,n,ier,band,flag,limit;
  double f,est,eps,upe,une,ydes;
  double x[33],g[33],h[625],w[100],y[100];
  double dmax();
  void inside(),roots(),funct(),dfmfp();
  FILE  *fp;
  fp = fopen(fname,"r");
  if (fp == NULL)
    { printf("cann't open this file\n"); exit(0); }
  fscanf(fp,"%d",&band);
  if (band <= 0)
    { m = 0;
      do
        { m++;
          fscanf(fp,"%lf%lf",&w[m],&y[m]);
        } while( w[m] < 0.5 );
    }
  else
    { m1 = 0;
      for (j=0;j<band;j++)
        { fscanf(fp,"%lf%lf%lf",&upe,&une,&ydes);
          m = m1;
          do
```

```
            { m++;
              w[m] = upe + ( (une-upe) * (m-m1-1) )/20.0;
              y[m] = ydes;
            } while( (m-m1)<=20 );
          m1 = m;
        }
    }
  fscanf(fp,"%d%d%lf%lf",&n,&limit,&est,&eps);
  fclose(fp);
  for (j=1;j<=624;j++)
    { h[j] = 0.0; }
  for (j=1;j<=32;j++)
    { g[j] = 0.0;
      x[j] = 0.0;
    }
  x[4] = 0.25;
  f = 0.0;
  ier = 0;
  call = 0;
  do
    { dfmfp(&n,x,&f,g,&est,&eps,25,&ier,h,w,y,m);
      inside(&n,x,&flag);
    } while( ( (flag ! = 0) || (ier ! = 0) ) && (call <= limit) );
  call = -10;
  funct(&n,x,&f,g,w,y,m);
}

static void funct(pn,x,pf,g,w,y,m)
int m, *pn;
double *pf,x[],g[],w[],y[];
{ int i,j,k,ibc,isav,j4,j41,order;
  double a,a1,a2,pi,attn,freq,ripp,une,upe,ydes;
  double qr,qi,nuder,nudei,qbarr,qbari,zcurr,zcuri,zcur2r,zcur2i;
  double yht[100],e[100];
  static double zr[100],zi[100];
  double c[40],d[40],am[300],ph[300];
  double numr[100][9],numi[100][9];
  double denr[100][9],deni[100][9];
```

```
char fname[40];
FILE *fp1;
pi = -4.0*atan(1.0);
k = *pn/4;
if (call == 0)
  { for (i=1;i<=m;i++)
      { zr[i] = cos(w[i]*2*pi);
        zi[i] = sin(w[i]*2*pi);
      }
  }
a1 = 0.0;
a2 = 0.0;
for(i=1;i<=m;i++)
  { zcurr = zr[i];
    zcuri = zi[i];
    zcur2r = zcurr*zcurr - zcuri*zcuri;
    zcur2i = 2*zcurr*zcuri;
    qr = 1.0;
    qi = 0.0;
    for (j=1;j<=k;j++)
      { j4 = (j-1)*4;
        numr[i][j] = x[j4+1]*zcurr + x[j4+2]*zcur2r + 1.0;
        numi[i][j] = x[j4+1]*zcuri + x[j4+2]*zcur2i;
        denr[i][j] = x[j4+3]*zcurr + x[j4+4]*zcur2r + 1.0;
        deni[i][j] = x[j4+3]*zcuri + x[j4+4]*zcur2i;
        nuder=numr[i][j]*denr[i][j]-numi[i][j]*deni[i][j];
        nuder=nuder/( pow(denr[i][j],2) + pow(deni[i][j],2) );
        nudei=numi[i][j]*denr[i][j]-numr[i][j]*deni[i][j];
        nudei = nudei/( pow(denr[i][j],2) + pow(deni[i][j],2) );
        qbarr = qr;
        qbari = qi;
        qr = qbarr*nuder - qbari*nudei;
        qi = qbari*nuder + qbarr*nudei;
      }
    qbarr = qr;
    qbari = -qi;
    yht[i] = qr*qr + qi*qi;
    a2 = a2 + yht[i];
```

```
      yht[i] = sqrt(yht[i]);
      a1 = a1 + yht[i] * y[i];
    }
  a = a1/a2;
  *pf = 0.0;
  for (j=1;j<=16;j++)
    { g[j] = 0.0; }
  for (i=1;i<=m;i++)
    { zcurr = zr[i];
      zcuri = zi[i];
      zcur2r = zcurr * zcurr - zcuri * zcuri;
      zcur2i = 2.0 * zcurr * zcuri;
      yht[i] = a * yht[i];
      e[i] = yht[i] - y[i];
      *pf = *pf + e[i] * e[i];
      for (j=1;j<=k;j++)
        { j4 = (j-1) * 4;
          nuder=numr[i][j]/(pow(numr[i][j],2)+pow(numi[i][j],2));
          nudei=-numi[i][j]/(pow(numr[i][j],2)+pow(numi[i][j],2));
          qr = 2.0 * e[i] * yht[i] * nuder;
          qi = 2.0 * e[i] * yht[i] * nudei;
          g[j4+1] += qr * zcurr - qi * zcuri;
          g[j4+2] += qr * zcur2r - qi * zcur2i;
          nuder=denr[i][j]/( pow(denr[i][j],2)+pow(deni[i][j],2));
          nudei=-deni[i][j]/(pow(denr[i][j],2)+pow(deni[i][j],2));
          qr = -2.0 * e[i] * yht[i] * nuder;
          qi = -2.0 * e[i] * yht[i] * nudei;
          g[j4+3] += qr * zcurr - qi * zcuri;
          g[j4+4] += qr * zcur2r - qi * zcur2i;
        }
    }
  call++;
  if ( call > 0 ) return;
  order = *pn/2;
  printf("\n\n");
  printf("           Infinite Impulse Response (IIR)\n");
  printf("           Fletcher-Powell Optimization Algorithm\n");
  printf("           Recursive Filter of Order %4d",order);
```

```
ibc = 1;
do
  { une = w[ibc];
    ydes = y[ibc];
    ibc++;
    for (i=ibc;i<=m;i++)
      { if( y[i] == ydes ) continue;
        upe = w[i-1];
        break;
      }
    if ( (i-1) == m ) upe = w[m];
    printf("\nBand edges are %5.2f         %5.2f         ",une,upe);
    printf("Desired value = %5.2f",ydes);
    ibc = i;
  } while ( ibc < m );
printf("\nFinal error function value = %15.8f", *pf);
attn = 0.0;
ripp = 0.0;
for (i=1;i<=m;i++)
  { if ( y[i] != 1.0 )
      { if ( (y[i] == 0.0) && (fabs(e[i]) >= attn) )
            attn = fabs(e[i]);
      }
    else
      { if (fabs(e[i]) >= ripp)
            ripp = fabs(e[i]);
      }
  }
ripp = 20.0*log10(1.0 + ripp);
attn = 20.0*log10(attn);
printf("\nMaximum absolute deviation in passband = %10.5lfdb\n",ripp);
printf("\nMinimum stopband attenuation = %10.5lfdb\n",attn);
printf("\nConstant multipler c* = %15.8f\n\n",a);
printf("Coefficients for cascade decomposition");
for (j=0;j<k;j++)
  { j4 = j*4;
    j41 = j4/4 + 1;
    printf("\n\nSection %1d",j+1);
```

```
printf("\n  a(%1d)=%15.8f     b(%1d)=%15.8f",j41,x[j4+1],j41,x[j4+2]);
printf("\n  c(%1d)=%15.8f     d(%1d)=%15.8f",j41,x[j4+3],j41,x[j4+4]);
        }
      isav = call;
      call = -10;
      roots(pn,x);
      call = isav;
      for (j=0;j<k;j++)
        { if (j == 0)
            { d[0] = a;
              d[1] = a * x[1];
              d[2] = a * x[2];
            }
          else
            { d[j * 3+0] = 1.0;
              d[j * 3+1] = x[4 * j+1];
              d[j * 3+2] = x[4 * j+2];
            }
          c[j * 3+0] = 1.0;
          c[j * 3+1] = x[4 * j+3];
          c[j * 3+2] = x[4 * j+4];
        }
      printf("\nenter file name of magnitude response\n");
      scanf("%s",fname);
      if ((fp1=fopen(fname,"w"))==NULL)
        { printf("cannot open file %s\n",fname);
          exit(0);
        }
      gainc(d,c,2,k,am,ph,201,2);
      for (i=0;i<201;i++)
        { freq = i * 0.5/200.0;
          fprintf(fp1,"%lf     %lf\n",freq,am[i]);
        }
      fclose(fp1);
    }

    static void inside(pn,x,pflag)
    int * pn, * pflag;
```

```
double x[];
{ int i = -1;
  double b c,disc,r1,r2,dr1,dr2;
   *pflag = 0;
  while ( (i+2) <=  *pn )
    { i = i + 2;
      b = -0.5 * x[i];
      c = x[i+1];
      disc = b * b - c;
      if (disc > 0.0)
        { disc = sqrt(disc);
          r1 = b + disc;
          r2 = b - disc;
          dr1 = fabs(r1);
          dr2 = fabs(r2);
          if( (dr1 <= 1.0) && (dr2 <= 1.0) ) continue;
           *pflag = 1;
          if (dr1 > 1.0) r1 = 1.0/r1;
          if (dr2 > 1.0) r2 = 1.0/r2;
          x[i] = -1.0 * (r1 + r2);
          x[i+1] = r1 * r2;
        }
      else
        { if (c <= 1.0) continue;
           *pflag = 1;
          c = 1.0/c;
          x[i+1] = c;
          x[i] *= c;
        }
    }
}

static void roots(pn,x)
int *pn;
double x[];
{ int j = -1;
  double b,c,r1,r2,disc,discm;
  if (call == -10)
```

```
    { printf("\n\n\nRoots:\n                    ");
      printf("Real            Imag            Real            Imag\n");
    }
  while ( (j+2) <=  *pn )
    { j = j + 2;
      if ( j%4 == 1 ) { }
      if ( j%4 == 1 )
        { printf("Section %1d\n",(j/4)+1);
          printf("   zeros");
        }
      else
        { printf("   poles"); }
      b = -0.5 * x[j];
      c = x[j+1];
      disc = b * b - c;
      if (disc > 0.0)
        { disc = sqrt(disc);
          r1 = b + disc;
          r2 = b - disc;
          if (call == -10)
            printf("%15.8lf                    %15.8lf\n",r1,r2);
        }
      else
        { disc = sqrt(-1.0 * disc);
          discm = -1.0 * disc;
          if (call == -10)
            printf("%15.8lf%15.8lf%15.8lf%15.8lf\n",b,disc,b,discm);
        }
    }
}

static void dfmfp(pn,x,pf,g,pest,peps,limit,pier,h,ww,y,m)
int m,limit, * pn, * pier;
double x[],g[],h[],ww[],y[];
double  * pf, * pest, * peps;
{ int i,j,k,l,n2,n3,n31,nj,kl,kount;
  double t,w,z,dx,dy,fx,fy,alfa,ambda,dalfa,hnrm,gnrm,oldf;
```

```
  funct(pn,x,pf,g,ww,y,m);
  *pier = 0;
  kount = 0;
  n2 = *pn + *pn;
  n3 = n2 + *pn;
  n31 = n3 + 1;
l1:
  k = n31;
  for (j=1;j<= *pn;j++)
    { h[k] = 1.0;
      nj = *pn - j;
      if (nj <= 0) hreak;
      for (l=1;l<=nj;l++)
        { kl = k+l;
          h[kl] = 0.0;
        }
      k = kl + 1;
    }
  while ( 1 )
    { kount = kount + 1;
      oldf = *pf;
      for (j=1;j<= *pn;j++)
        { k = *pn + j;
          h[k] = g[j];
          k += *pn;
          h[k] = x[j];
          k = j - n3;
          t = 0.0;
          for (l=1;l<= *pn;l++)
            { t -= g[l]*h[k];
              if (l < j)
                k += *pn-l;
              else
                k++;
            }
          h[j] = t;
        }
      dy = 0.0;
```

```
        hnrm = 0.0;
        gnrm = 0.0;
        for (j=1;j<= *pn;j++)
          { hnrm += fabs(h[j]);
            gnrm += fabs(g[j]);
            dy += h[j] * g[j];
          }
        if (dy >= 0.0) goto l4;
        if( hnrm/gnrm <=  *peps ) goto l4;
        fy =  *pf;
        alfa = 2.0 * ( * pest -  * pf)/dy;
        ambda = 1.0;
        if (alfa > 0.0)
          { if (alfa < ambda) ambda = alfa; }
        alfa=0.0;
        do
          { fx = fy;
            dx = dy;
            for (i=1;i<= *pn;i++)
              { x[i] = x[i] + ambda * h[i]; }
            funct(pn,x,pf,g,ww,y,m);
            fy =  *pf;
            dy = 0.0;
            for (i=1;i<= *pn;i++)
              { dy = dy + g[i] * h[i]; }
            if (dy == 0.0) goto l3;
            if ( ( dy > 0.0) || ( fy >= fx ) ) goto l2;
            ambda += alfa;
            alfa = ambda;
          } while ( hnrm * ambda <= 1.0 );
         * pier = 2;
        return;
l2:
        while ( 1 )
          { t = 0.0;
            while ( 1 )
              { if (ambda == 0.0) goto l3;
                z = 3.0 * (fx-fy)/ambda + dx + dy;
```

```
            alfa = dmax(fabs(z),fabs(dx),fabs(dy));
            dalfa = z/alfa;
            dalfa = pow(dalfa,2) - (dx/alfa) * (dy/alfa);
            if ( dalfa < 0.0 ) goto l4;
            w = alfa * sqrt(dalfa);
            alfa = dy - dx + 2 * w;
            if (alfa ! = 0.0)
              { alfa = (dy - z + w)/alfa; }
            else
              { alfa = (z + dy - w)/(z + dx + z + dy); }
            alfa = alfa * ambda;
            for (i=1;i<= * pn;i++)
              { x[i] = x[i] + (t-alfa) * h[i]; }
            funct(pn,x,pf,g,ww,y,m);
            if ( * pf <= fx)
              { if ( * pf <= fy) goto l3; }
            dalfa = 0.0;
            for (i=1;i<= * pn;i++)
              { dalfa = dalfa + g[i] * h[i]; }
            if ( (dalfa >= 0.0) || ( * pf > fx) ) break;
            if ( ( * pf == fx) && (dx == dalfa) ) goto l3;
            fx = * pf;
            dx = dalfa;
            t = alfa;
            ambda = alfa;
          }
        if (fy == * pf)
          { if (dy == dalfa) goto l3; }
        fy = * pf;
        dy = dalfa;
        ambda = ambda - alfa;
      }
l3:
    if ( (oldf - * pf + * peps) < 0.0 ) goto l4;
    for (j=1;j<= * pn;j++)
      { k = * pn + j;
        h[k] = g[j] - h[k];
        k = * pn + k;
```

```
      h[k] = x[j] - h[k];
    }
  *pier = 0;
  if (kount >= *pn)
    { t = 0.0;
      z = 0.0;
      for (j=1;j<= *pn;j++)
        { k = *pn + j;
          w = h[k];
          k = k + *pn;
          t = t + fabs(h[k]);
          z = z + w * h[k];
        }
      if ( (hnrm <= *peps) && (t <= *peps) ) return;
    }
  if (kount >= limit)
    { *pier = 1;
      return;
    }
  alfa = 0.0;
  for (j=1;j<= *pn;j++)
    { k = j + n3;
      w = 0.0;
      for (l=1;l<= *pn;l++)
        { kl = *pn + l;
          w = w + h[kl] * h[k];
          if (l < j)
            { k += *pn - l; }
          else
            { k = k + 1; }
        }
      k = *pn + j;
      alfa = alfa + w * h[k];
      h[j] = w;
    }
  if (z * alfa == 0.0) goto l1;
  k = n31;
  for (l=1;l<= *pn;l++)
```

```
        { kl = n2 + l;
          for (j=l;j<= *pn;j++)
            { nj = n2 + j;
              h[k] = h[k] + h[kl] * h[nj]/z - h[l] * h[j]/alfa;
              k = k + 1;
            }
        }
    }
   *pier = 1;
   return;
 l4:
   for (j=1;j<= *pn;j++)
     { k = n2 + j;
       x[j] = h[k];
     }
   funct(pn,x,pf,g,ww,y,m);
   if ( gnrm >  *peps )
     { if ( *pier >= 0.0)
         { *pier = -1;
           goto l1;
         }
     }
   else
     { *pier = 0; }
 }

 static double dmax(a,b,c)
 double a,b,c;
 { double d;
   d = (a <= b) ? b : a;
   if (d <= c) d = c;
   return(d);
 }
```

五、例　题

数字滤波器的技术指标一般有两种形式：一种是给定若干离散频率点及其所对应的幅值；另一种是给定滤波器的通带和阻带范围。用本节方法设计数字滤波器时，要将滤波器的技术指标建立一个数据文件。在数据文件中，首先给定参数band。若band≤0，则

表示频率点输入方式；否则为频带输入方式，band 表示频带数；其次，给定若干离散频率点及其所对应的幅值，或者给定滤波器的通带和阻带范围；最后，给定四个有关参数：滤波器系数的个数 n，函数计算次数极限 limit，函数最小估计值 est，以及所要求的绝对误差值 eps。具体使用方法见下面的例子。

主函数仅要求输入数字滤波器技术指标的文件名。

主函数程序(文件名:iircad.m)：

```
#include "stdio.h"
#include "iircad.c"
main()
{ char fname[40];
  printf("\nenter data file name\n");
  scanf("%s",fname);
  iircad(fname);
}
```

例 1：设计 IIR 低通数字滤波器，其技术指标为

$$
H_d(f)=\begin{cases}1.0 & f=0.0,0.005,0.01,0.015,0.02,\cdots,0.045\\ 0.5 & f=0.05\\ 0.0 & f=0.055,0.06,0.065,0.07,\cdots,0.095\\ 0.0 & f=0.1,0.15,0.2,0.25,\cdots,0.5\end{cases}
$$

选择参数 n=8，limit=500，est=0.0，eps=0.00001。用频率点输入方式，建立数据文件(文件名:iirp.dat)如下：

```
0
0.000   1.0
0.005   1.0
0.010   1.0
0.015   1.0
0.020   1.0
  ⋮      ⋮
0.045   1.0
0.050   0.5
0.055   0.0
0.060   0.0
0.065   0.0
0.070   0.0
  ⋮      ⋮
0.095   0.0
0.100   0.0
0.150   0.0
```

```
0.200   0.0
0.250   0.0
  ⋮      ⋮
0.500   0.0
8   500   0.0   0.00001
```

运行结果：

```
            Infinite Impulse Response (IIR)
            Fletcher—Powell Optimization Algorithm
            Recursive Filter of Order      4
Band edges are  0.00        0.04       Desired value = 1.00
Band edges are  0.05        0.05       Desired value = 0.50
Band edges are  0.06        0.50       Desired value = 0.00
Final error function value =          0.03395907
Maximum absolute deviation in passband =       0.30519 dB
Minimum stopband attenuation = -17.96839 dB
Constant multipler c * =       0.02486738
Coefficients for cascade decomposition
Section 1
  a(1)=      -1.22274366     b(1)=      0.99999998
  c(1)=      -1.72909447     d(1)=      0.76527448
Section 2
  a(2)=      -1.85076922     b(2)=      0.99999998
  c(2)=      -1.86243676     d(2)=      0.94400191
Roots:
                 Real          Imag          Real          Imag
Section 1
  zeros      0.61137183    0.79134346    0.61137183    -0.79134346
  poles      0.86454723    0.13353860    0.86454723    -0.13353860
Section 2
  zeros      0.92538461    0.37902943    0.92538461    -0.37902943
  poles      0.93121838    0.27718988    0.93121838    -0.27718988
```

该数字滤波器的幅频响应如图 2—5—13 所示。

例 2：设计 IIR 低通数字滤波器，其技术指标为

$$H_d(f) = \begin{cases} 1.0 & 0 \leqslant f \leqslant 0.16 \\ 0.0 & 0.2 \leqslant f \leqslant 0.5 \end{cases}$$

选择参数 n=8，limit=500，est=0.0，eps=0.00001。用频带输入方式，建立数据文件(文件名：iirb.dat)如下：

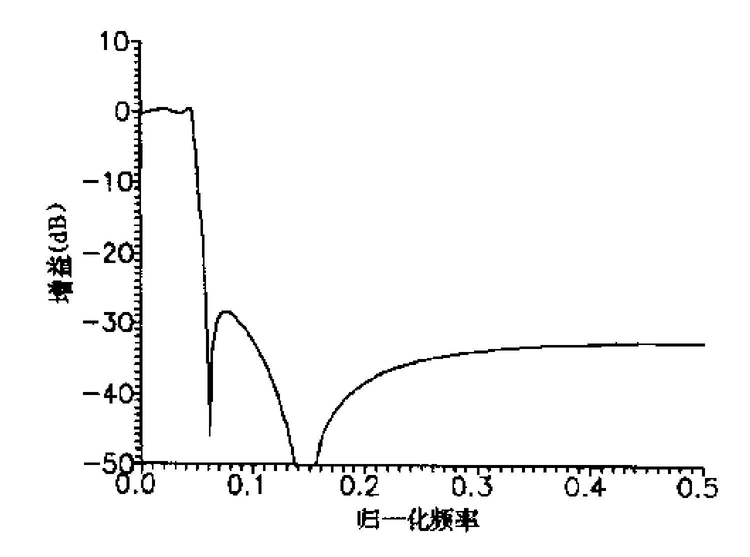


图 2—5—13　低通 IIR 数字滤波器的幅频响应(例 1)

```
2
0.0     0.16    1.0
0.2     0.5     0.0
8  500  0.0  0.00001
```

运行结果：

```
              Infinite Impulse Response (IIR)
              Fletcher—Powell Optimization Algorithm
              Recursive Filter of Order     4
Band edges are  0.00       0.16      Desired value =    1.00
Band edges are  0.20       0.50      Desired value =    0.00
Final error function value =          0.03020483
Maximum absolute deviation in passband =      0.38934 dB
Minimum stopband attenuation =    −19.43039 dB
Constant multipler c * =          0.05905215
Coefficients for cascade decomposition
Section 1
  a(1)=        0.19750014    b(1)=        0.99999996
  c(1)=       −1.03898522    d(1)=        0.37798107
Section 2
  a(2)=        0.19750014    b(2)=        0.99999996
  c(2)=       −0.94936040    d(2)=        0.80729536
Roots:
               Real          Imag           Real          Imag
Section 1
  zeros     −0.09875007    0.99511225    −0.09875007    −0.99511225
```

poles	0.51949261	0.32879857	0.51949261	−0.32879857
Section 2				
zeros	−0.09875007	0.99511225	−0.09875007	−0.99511225
poles	0.47468020	0.76287224	0.47468020	−0.76287224

该数字滤波器的幅频响应如图 2—5—14 所示。

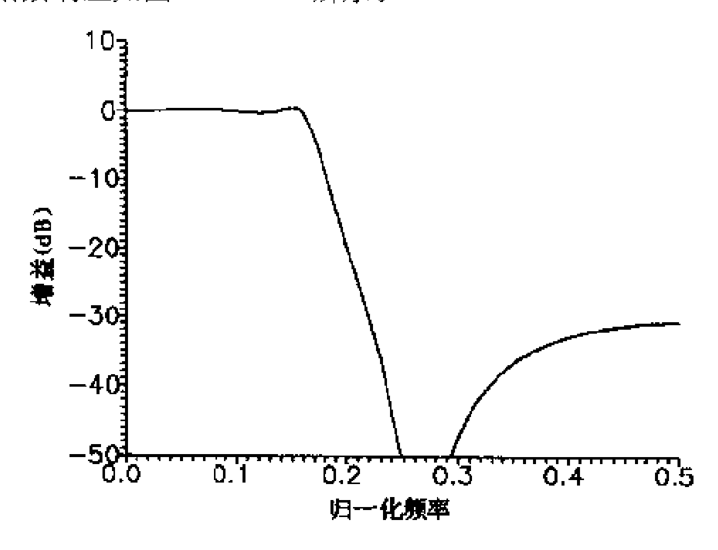


图 2—5—14 低通 IIR 数字滤波器的幅频响应(例 2)

第六章　FIR 数字滤波器的设计

§6.1　窗函数方法

一、功能

用窗函数方法设计线性相位 FIR 数字滤波器。

二、方法简介

设 $N-1$ 阶 FIR 数字滤波器的单位冲激响应为 $h(n)$，则传递函数 $H(z)$ 为

$$H(z) = \sum_{n=0}^{N-1} h(n) z^{-n}$$

窗函数法的设计步骤如下：

1. 根据给定的理想频率响应 $H_d(e^{jw})$，利用傅立叶反变换，求出单位冲激响应 $h_d(n)$

$$h_d(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_d(e^{jw}) e^{jwn} dw$$

2. 将 $h_d(n)$ 乘以窗函数 $w(n)$，得到所要求的 FIR 滤波器系数 $h(n)$

$$h(n) = w(n) h_d(n) \quad , \quad 0 \leqslant n \leqslant N-1$$

常用的窗函数有：

矩形窗

$$w(n) = 1 \quad , \quad 0 \leqslant n \leqslant N-1$$

图基(Tukey)窗

$$w(n) = \begin{cases} 0.5\left(1-\cos\left(\dfrac{10\pi n}{N+8}\right)\right) & , \quad 0 \leqslant n \leqslant (N-2)/10 \\ 1 & , \quad (N-2)/10 \leqslant n \leqslant 9(N-2)/10 \\ 0.5\left(1-\cos\left(\dfrac{10\pi(N-n-1)}{N+8}\right)\right) & , \quad 9(N-2)/10 \leqslant n \leqslant N-1 \end{cases}$$

三角窗

$$w(n) = 1 - \left|1 - \frac{2n}{N-1}\right| \quad , \quad 0 \leqslant n \leqslant N-1$$

汉宁(Hanning)窗

$$w(n) = 0.5 - 0.5\cos\frac{2\pi n}{N-1} \quad , \quad 0 \leqslant n \leqslant N-1$$

海明(Hamming)窗

$$w(n) = 0.54 - 0.46\cos\frac{2\pi n}{N-1} \quad , \quad 0 \leqslant n \leqslant N-1$$

布拉克曼(Blackman)窗

$$w(n) = 0.42 - 0.5\cos\frac{2\pi n}{N-1} + 0.08\cos\frac{4\pi n}{N-1} \quad , \quad 0 \leqslant n \leqslant N-1$$

凯塞(Kaiser)窗

$$w(n) = \frac{I_0(\beta\sqrt{1-(1-2n/(N-1))^2})}{I_0(\beta)} \quad , \quad 0 \leqslant n \leqslant N-1$$

其中 $I_0(\beta)$是第一类零阶修正贝塞耳函数。β是控制窗函数形状的参数，β越大，$w(n)$窗越窄，频谱的旁瓣越小，但主瓣也相应加宽。β的典型值为 $4 \leqslant \beta \leqslant 9$。$\beta=0$ 时，凯塞窗变为矩形窗；$\beta=5.44$ 时，凯塞窗与海明窗接近；$\beta=8.5$ 时，凯塞窗与布拉克曼窗接近。

三、使用说明

1. 子函数语句

void firwin(n,band,fln,fhn,wn,h)

2. 形参说明

n ——整型变量。滤波器的阶数。

band ——整型变量。滤波器的类型。取值为 1、2、3 和 4，分别对应低通、高通、带通和带阻滤波器。

fln ——双精度实型变量。

fhn ——双精度实型变量。

对于低通和高通滤波器，fln：通带边界频率：对于带通和带阻滤波器，fln：通带下边界频率，fhn：通带上边界频率

wn ——整型变量。窗函数的类型；取值 1 到 7，分别对应矩形窗，图基窗，三角窗，汉宁窗，海明窗，布拉克曼窗和凯塞窗。

h ——双精度实型一维数组，长度为(n+1)。存放 FIR 滤波器的系数。

函数 firwin() 调用以下函数：

子函数 window()，窗函数的计算。

子函数 Kaiser()，凯塞窗的计算。

子函数 bessel0()，贝塞耳函数的计算。

四、子函数程序(文件名：firwin.c)

```
#include "math.h"
void firwin(n,band,fln,fhn,wn,h)
int n,band,wn;
double fln,fhn,h[];
 { int i,n2,mid;
```

```
double s,pi,wc1,wc2,beta,delay;
double window( );
beta = 0.0;
if (wn == 7)
  {printf("input beta parameter of Kaiser window ( 3 < beta < 10 )\n");
    scanf("%lf",&beta);
  }
pi = 4.0 * atan(1.0);
if ( (n%2) == 0 )
  { n2 = n/2 - 1;
    mid = 1;
  }
else
  { n2 = n/2;
    mid = 0;
  }
delay = n/2.0;
wc1 = 2.0 * pi * fln;
if ( band >= 3 ) wc2 = 2.0 * pi * fhn;
switch ( band )
  { case 1:
      { for (i=0;i<=n2;i++)
          { s = i - delay;
            h[i] = ( sin(wc1 * s)/(pi * s)) * window(wn,n+1,i,beta);
            h[n-i] = h[i];
          }
        if ( mid == 1 ) h[n/2] = wc1/pi;
        break;
      }
    case 2:
      { for (i=0; i<=n2; i++)
          { s = i - delay;
            h[i] = (sin(pi * s)-sin(wc1 * s))/(pi * s);
            h[i] = h[i] * window(wn,n+1,i,beta);
            h[n-i] = h[i];
          }
        if ( mid == 1 ) h[n/2] = 1.0 - wc1/pi;
        break;
```

```
         }
      case 3:
         { for (i=0; i<=n2; i++)
            { s = i - delay;
              h[i] = (sin(wc2 * s)-sin(wc1 * s))/(pi * s);
              h[i] = h[i] * window(wn,n+1,i,beta);
              h[n-i] = h[i];
            }
           if ( mid == 1 ) h[n/2] = (wc2-wc1)/pi;
           break;
         }
      case 4:
         { for (i=0; i<=n2; i++)
            { s = i - delay;
              h[i] = (sin(wc1 * s)+sin(pi * s)-sin(wc2 * s))/(pi * s);
              h[i] = h[i] * window(wn,n+1,i,beta);
              h[n-i] = h[i];
            }
           if ( mid == 1 ) h[n/2] = (wc1+pi-wc2)/pi;
           break;
         }
    }
}

static double window(type,n,i,beta)
int i,n,type;
double beta;
{ int k;
  double pi,w;
  double kaiser();
  pi = 4.0 * atan(1.0);
  w = 1.0;
  switch ( type )
    { case 1:
        { w = 1.0 ;
          break;
        }
      case 2:
```

```
        { k = (n-2)/10;
          if ( i <= k )
            w = 0.5 * (1.0-cos(i * pi/(k+1)));
          if (i > n-k-2)
            w = 0.5 * (1.0-cos((n-i-1) * pi/(k+1)));
          break;
        }
      case 3:
        { w = 1.0 - fabs(1.0-2 * i/(n-1.0));
          break;
        }
      case 4:
        { w = 0.5 * (1.0 - cos(2 * i * pi/(n-1)));
          break;
        }
      case 5:
        { w = 0.54 - 0.46 * cos(2 * i * pi/(n-1));
          break;
        }
      case 6:
        { w=0.42-0.5 * cos(2 * i * pi/(n-1))+0.08 * cos(4 * i * pi/(n-1));
          break;
        }
      case 7:
        { w = kaiser(i,n,beta);
          break;
        }
    }
  return(w);
}

static double kaiser(i,n,beta)
int i,n;
double beta;
{ double a,w,a2,b1,b2,beta1;
  double bessel0();
  b1 = bessel0(beta);
  a = 2.0 * i/(double)(n-1) - 1.0;
```

```
  a2 = a * a;
  beta1 = beta * sqrt(1.0 - a2);
  b2 = bessel0(beta1);
  w = b2/b1;
  return(w);
}

static double bessel0(x)
double x;
{ int i;
  double d,y,d2,sum;
  y = x/2.0;
  d = 1.0;
  sum = 1.0;
  for (i=1;i<=25;i++)
    { d = d * y/i;
      d2 = d * d;
      sum = sum + d2;
      if ( d2 < sum * (1.0e-8) ) break;
    }
  return(sum);
}
```

五、例 题

下面给出主函数程序，它调用 firwin.c 子函数。通过人机对话，它可以设计低通、高通、带通和带阻这四种形式的滤波器。下面对输入参数进行说明：

band：滤波器的类型；n：滤波器的阶数；fs：采样频率；对于低通和高通滤波器，fl：通带边界频率；对于带通和带阻滤波器，fl：通带下边界频率，fh：通带上边界频率；wn：窗函数的类型；fname：幅频响应文件名。

主函数要调用计算滤波器频率响应的函数 gain()，参看第二篇§4.1节。

主函数程序(文件名:firwin.m)：

```
#include "stdio.h"
#include "firwin.c"
#include "gain.c"
main( )
{ int i,j,n,n2,band,wn;
  double fl,fh,fs,freq;
```

```
double h[100],c[100],x[300],y[300];
char fname[40];
FILE *fp;
c[1]=0.0;
printf("select one of the four types for FIR digital filter\n");
printf(" 1 -- lowpass; 2 -- highpass\n");
printf(" 3 -- bandpass; 4 -- bandstop\n");
scanf("%d",&band);
printf("input the filter order n\n");
scanf("%d",&n);
printf("input low cutoff frequrncy fl\n");
scanf("%lf",&fl);
fh = 0.0;
if ( band >= 3 )
  { printf("input high cutoff frequency fh\n");
    scanf("%lf",&fh);
  }
printf("input sample frequence fs\n");
scanf("%lf",&fs);
printf("select window \n");
printf(" 1 -- rectangular; 2 -- tapered rectangular\n");
printf(" 3 -- triangular ; 4 -- Hanning\n");
printf(" 5 -- Hamming) ; 6 -- Blackman\n");
printf(" 7 -- Kaiser\n");
scanf("%d",&wn);
fl = fl/fs;
fh = fh/fs;
firwin(n,band,fl,fh,wn,h);
printf(" FIR digital filter\n");
printf(" * * * * impulse response * * * *\n\n");
n2 = n/2;
for (i=0;i<=n2;i++)
{ j = n - i;
    printf(" h(%2d) = %12.8lf = h(%2d)\n",i,h[i],j);
  }
printf("\ninput file name of frequency response\n");
scanf("%s",fname);
if ( (fp=fopen(fname,"w")) == NULL )
```

```
        { printf("cannot open this file\n");
          exit(1);
        }
    gain(h,c,n,1,x,y,300,2);
    for (i=0;i<300;i++)
        { freq = 0.5 * i/299;
          fprintf(fp," %lf %lf \n",freq,x[i]);
        }
    fclose(fp);
}
```

例 1：设计一个 30 阶的 FIR 低通数字滤波器，其通带边界频率为 0.2。选择参数 n=30，band=1，fl=0.2，fs=1，wn=1，采用矩形窗函数。

运行结果：

滤波器的系数为

```
h( 0) =   -0.00000000   = h(30)
h( 1) =   -0.02162362   = h(29)
h( 2) =   -0.01439214   = h(28)
h( 3) =    0.01559149   = h(27)
h( 4) =    0.02752097   = h(26)
h( 5) =   -0.00000000   = h(25)
h( 6) =   -0.03363674   = h(24)
h( 7) =   -0.02338723   = h(23)
h( 8) =    0.02672827   = h(22)
h( 9) =    0.05045512   = h(21)
h(10) =   -0.00000000   = h(20)
h(11) =   -0.07568267   = h(19)
h(12) =   -0.06236595   = h(18)
h(13) =    0.09354893   = h(17)
h(14) =    0.30273069   = h(16)
h(15) =    0.40000000   = h(15)
```

滤波器的幅频响应如图 2—6—1 所示。

例 2：设计一个 30 阶的 FIR 高通数字滤波器，其通带边界频率为 0.3。选择参数 n=30，band=2，fl=0.3，fs=1，wn=4，采用海明窗函数。

运行结果：

滤波器的系数为

```
h( 0) =    0.00000000   = h(30)
h( 1) =   -0.00023626   = h(29)
h( 2) =    0.00062213   = h(28)
```

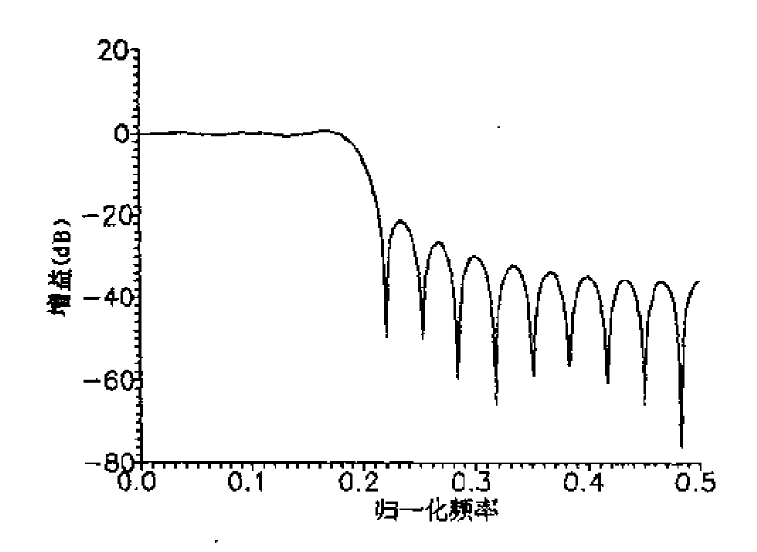


图 2—6—1　低通 FIR 数字滤波器的幅频响应

h(3) = 0.00148885 = h(27)
h(4) = −0.00455292 = h(26)
h(5) = −0.00000000 = h(25)
h(6) = 0.01162121 = h(24)
h(7) = −0.01047130 = h(23)
h(8) = −0.01476106 = h(22)
h(9) = 0.03302330 = h(21)
h(10) = 0.00000000 = h(20)
h(11) = −0.06316213 = h(19)
h(12) = 0.05641053 = h(18)
h(13) = 0.08950506 = h(17)
h(14) = −0.29942300 = h(16)
h(15) = 0.40000000 = h(15)

滤波器的幅频响应如图 2—6—2 所示。

例 3：设计一个 35 阶的 FIR 带通数字滤波器，其通带下边界频率为 0.2，通带上边界频率为 0.35。选择参数 n=35，band=3，fl=0.2，fh=0.35，fs=1，wn=7，β=4，采用凯塞窗函数。

运行结果：

滤波器的系数为

h(0) = 0.00113801 = h(35)
h(1) = −0.00495031 = h(34)
h(2) = −0.00050242 = h(33)
h(3) = 0.00547601 = h(32)

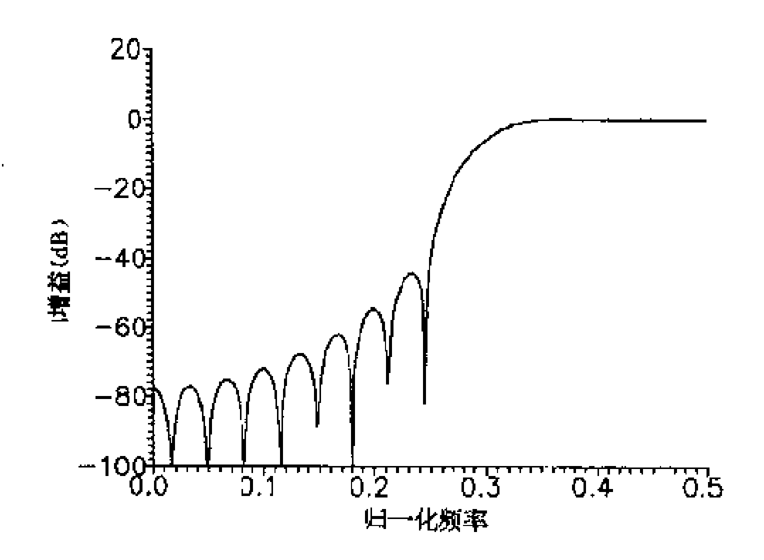


图 2—6—2　高通 FIR 数字滤波器的幅频响应

h(4) = −0.00026031 = h(31)
h(5) = 0.00662038 = h(30)
h(6) = −0.00961510 = h(29)
h(7) = −0.02279672 = h(28)
h(8) = 0.02875198 = h(27)
h(9) = 0.01936872 = h(26)
h(10) = −0.02157100 = h(25)
h(11) = 0.00140347 = h(24)
h(12) = −0.05065316 = h(23)
h(13) = 0.00842925 = h(22)
h(14) = 0.16445615 = h(21)
h(15) = −0.08690071 = h(20)
h(16) = −0.23204926 = h(19)
h(17) = 0.19276465 = h(18)

滤波器的幅频响应如图 2—6—3 所示。

例 4：设计一个 40 阶的 FIR 带阻数字滤波器，其通带下边界频率为 0.1，通带上边界频率为 0.25。选择参数 n=40，band=4，fl=0.1，fh=0.25，fs=1，wn=5，采用海明窗函数。

运行结果：

滤波器的系数为

h(0) = −0.00000000 = h(40)
h(1) = 0.00059158 = h(39)
h(2) = −0.00172412 = h(38)

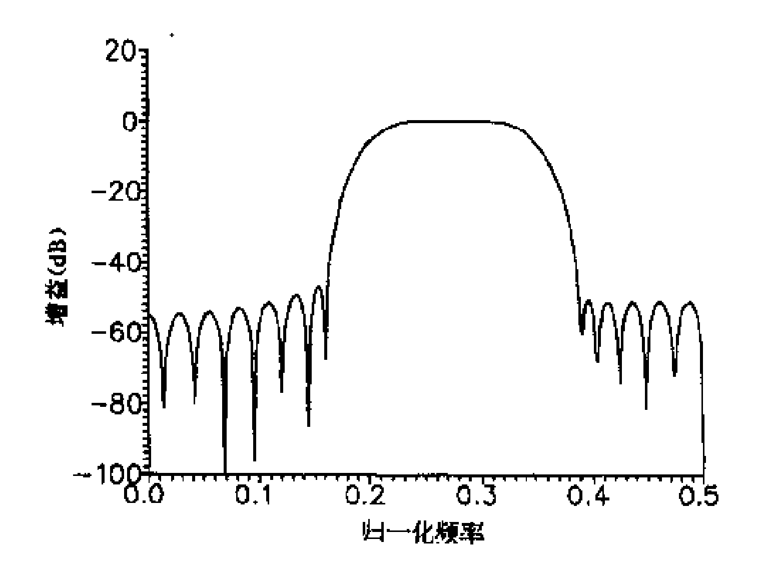


图 2—6—3 带通 FIR 数字滤波器的幅频响应

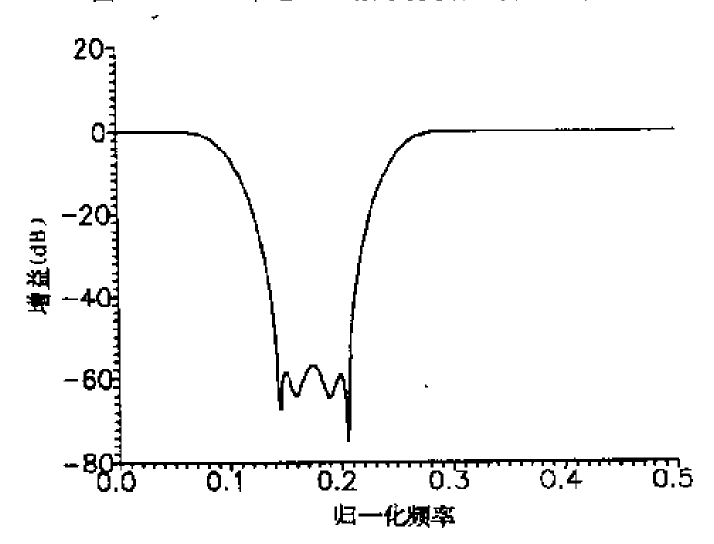


图 2—6—4 带阻 FIR 数字滤波器的幅频响应

h(3) = −0.00475414 = h(37)
h(4) = −0.00196280 = h(36)
h(5) = 0.00455673 = h(35)
h(6) = 0.00360322 = h(34)
h(7) = −0.00039687 = h(33)
h(8) = 0.01003684 = h(32)
h(9) = 0.02150465 = h(31)
h(10) = −0.00000000 = h(30)
h(11) = −0.03436547 = h(29)
h(12) = −0.02581339 = h(28)
h(13) = 0.00166661 = h(27)
h(14) = −0.02527010 = h(26)

h(15) = −0.05508474 = h(25)
h(16) = 0.04266523 = h(24)
h(17) = 0.19663449 = h(23)
h(18) = 0.14795751 = h(22)
h(19) = −0.13046893 = h(21)
h(20) = 0.70000000 = h(20)

滤波器的幅频响应如图 2－6－4 所示。

§6.2 频域最小误差平方设计

一、功能

用频域最小误差平方方法设计线性相位 FIR 低通数字滤波器。

二、方法简介

设 $N-1$ 阶 FIR 数字滤波器的单位冲激响应为 h(n)，则传递函数 H(z)为

$$H(z) = \sum_{n=0}^{N-1} h(n) z^{-n}$$

FIR 数字滤波器的实际幅度响应 H(ω)与理想幅度响应 $H_d(\omega)$的误差函数定义为

$$E = \frac{1}{2\pi}\int_{-\pi}^{\pi} |H(\omega) - H_d(\omega)|^2 d\omega$$

根据帕塞瓦定理，误差函数可以表示为

$$E = \sum_{n=-\infty}^{+\infty} |h(n) - h_d(n)|^2$$

$$= \sum_{n=-M}^{M} |h(n) - h_d(n)|^2 + \sum_{n=M+1}^{\infty} 2h_d^2(n)$$

其中 $M=(N-1)/2$。该式表明，若要使 E 最小，必须选择 $h(n)$，使它与 $h_d(n)$对应的 N 个值相等，即 $h(n)=h_d(n)$，$-M\leqslant n\leqslant M$

为了减少吉伯斯(Gibbs)效应，通常在通带与阻带之间加一个过渡带($f_c\leqslant f\leqslant f_s$)。过渡带函数常采用 P 阶样条函数，此时滤波器的系数为

$$h(n)=\left\{\frac{\sin[\pi(f_s-f_c)(n-M)/P]}{\pi(f_s-f_c)(n-M)/P}\right\}^P \frac{\sin[\pi(f_s+f_c)(n-M)]}{\pi(n-M)}, \quad 0\leqslant n\leqslant N-1$$

过渡带函数也可采用升余弦函数，此时滤波器的系数为

$$h(n)=\frac{\cos[\pi(f_s-f_c)(n-M)]}{1-4(f_s-f_c)^2(n-M)^2}\frac{\sin[\pi(f_s+f_c)(n-M)]}{\pi(n-M)}, \quad 0\leqslant n\leqslant N-1$$

三、使用说明

1. 子函数语句

void firls(n,fc,fs,tp,h)

2 形参说明

n ——整型变量。滤波器的阶数。

fc ——双精度实型变量。通带边界频率。

fs ——双精度实型变量。阻带边界频率。

tp ——整型变量。过渡带函数的类型。tp=0，表示升余弦函数；tp≥1，表示 tp 阶样条函数。

h ——双精度实型一维数组，长度为(n+1)。存放 FIR 滤波器的系数。

四、子函数程序(文件名:firls.c)

```
#include "math.h"
void firls(n,fc,fs,tp,h)
int n,tp;
double fc,fs,h[];
{ double fq,fr;
  void ls(),wgt();
  fq = fs - fc;
  fr = fs + fc;
  ls(h,n,fr);
  wgt(h,n,tp,fq);
}

static void ls(h,n,fc)
int n;
double fc,h[];
{ int i,m,n2;
  double q,am,pi;
  pi = 4.0*atan(1.0);
  m = n/2;
  am = n/2.0;
  n2 = (n-1)/2;
  if ( m == am ) h[m]=fc;
  for (i=0;i<=n2;i++)
    { q = pi*(i-am);
      h[i] = sin(fc*q)/q;
    }
}

static void wgt(h,n,tp,fq)
```

```
int n,tp;
double fq,h[];
{ int i;
  double q,am,pi,q1,wt;
  pi = 4.0 * atan(1.0);
  q = pi * fq;
  am = n/2.0;
  if (fq == 0.0 ) return;
  if (tp != 0)
    { for (i=0;i<am;i++)
      { q1 = q * (i-am)/tp;
        wt = pow(sin(q1)/q1,tp);
        h[i] = wt * h[i];
      }
    }
  else
    { for (i=0;i<am;i++)
      { wt = cos(q * (i-am));
        if ( fabs(wt) > 1.0e-6 )
          { wt =wt/(1 - pow((2 * fq * (i-am)),2)); }
        else
          { wt = pi/4.0; }
        h[i] = wt * h[i];
      }
    }
  for (i=0;i<am;i++)
    { h[n-i] = h[i]; }
```

五、例　题

下面给出主函数程序，它调用 firls.c 子函数。通过人机对话，它可以设计线性相位FIR 低通数字滤波器。下面对输入参数进行说明：

n：滤波器的阶数；f：采样频率；fc：通带边界频率；fs：阻带边界频率；tp：样条函数的阶数；fname：幅频响应文件名。

主函数要调用计算滤波器频率响应的函数 gain()，参看第二篇 §4.1 节。

主函数程序(文件名:firls.m)：

```
#include "stdio.h"
#include "firls.c"
#include "gain.c"
```

```
main()
{ int i,j,n,n2,tp;
  double f,fc,fs,freq;
  double h[100],c[100],x[300],y[300];
  char fname[40];
  FILE *fp;
  c[1]=0.0;
  printf("input the filter order n\n");
  scanf("%d",&n);
  printf("input passband cutoff frequrncy fc\n");
  scanf("%lf",&fc);
  printf("input stopband edge frequrncy fs\n");
  scanf("%lf",&fs);
  printf("input sample frequrncy f\n");
  scanf("%lf",&f);
  fc = fc/f;
  fs = fs/f;
  printf("input the order of spline n\n");
  scanf("%d",&tp);
  firls(n,fc,fs,tp,h);
  printf("                    FIR digital filter\n");
  printf("               * * * * impulse response * * * *\n\n");
  n2 = n/2;
  for (i=0;i<=n2;i++)
    { j = n - i;
      printf("         h(%2d) = %12.8lf = h(%2d)\n",i,h[i],j);
    }
  printf("\ninput file name of frequence response\n");
  scanf("%s",fname);
  if ( (fp=fopen(fname,"w")) == NULL )
    { printf("cannot open this file\n");
      exit(1);
    }
  gain(h,c,n,1,x,y,300,2);
  for (i=0;i<300;i++)
    { freq = 0.5*i/299;
      fprintf(fp,"%lf     %lf \n",freq,x[i]);
    }
```

```
    fclose(fp);
}
```

例:设计一个 20 阶的 FIR 低通数字滤波器,其通带边界频率为 0.2,阻带边界频率为 0.3,采用一阶样条函数作为过渡带。选择参数 n=20, fc=0.2, fs=0.3, f=1, ιp=1。

运行结果:

滤波器的系数为

```
h( 0) =   0.00000000   = h(20)
h( 1) =   0.00386543   = h(19)
h( 2) = -0.00000000   = h(18)
h( 3) = -0.01672869   = h(17)
h( 4) =   0.00000000   = h(16)
h( 5) =   0.04052847   = h(15)
h( 6) = -0.00000000   = h(14)
h( 7) = -0.09107840   = h(13)
h( 8) =   0.00000000   = h(12)
h( 9) =   0.31309968   = h(11)
h(10) =   0.50000000   = h(10)
```

滤波器的幅频响应如图 2—6—5 所示。

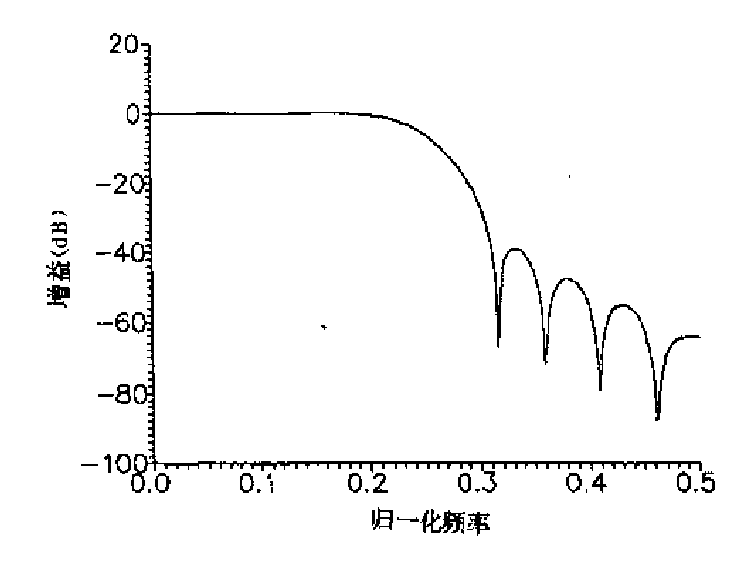


图 2—6—5 低通 FIR 数字滤波器的幅频响应

§6.3 切比雪夫逼近方法

一、功能

用切比雪夫最大误差最小化准则,设计线性相位 FIR 多带数字滤波器、微分器和希

尔伯特变换器。该方法也称为 Parks-McClellan 算法。

二、方法简介

设 $N-1$ 阶 FIR 数字滤波器的单位冲激响应为 $h(n)$，其传递函数 $H(z)$ 为

$$H(z)=\sum_{n=0}^{N-1}h(n)z^{-n}$$

设计线性相位 FIR 滤波器的切比雪夫逼近方法由以下步骤组成：

1. FIR 数字滤波器的频率响应

线性相位 FIR 数字滤波器的频率响应为

$$H(e^{j\omega})=H(\omega)\,e^{j\left(\frac{L\pi}{2}-\frac{N-1}{2}\omega\right)}$$

式中实函数 $H(\omega)$是滤波器的幅频响应。$L=0$ 或 1。当 $h(n)$为偶对称时，$L=0$；$h(n)$为奇对称时，$L=1$。

四种情况的幅频响应 $H(\omega)$为

情况 1：$h(n)$为偶对称，N 为奇数

$$H(\omega)=\sum_{n=0}^{(N-1)/2}a(n)\cos(n\omega)$$

情况 2：$h(n)$为偶对称，N 为偶数

$$H(\omega)=\sum_{n=1}^{N/2}b(n)\cos[(n-\frac{1}{2})\omega]$$

情况 3：$h(n)$为奇对称，N 为奇数

$$H(\omega)=\sum_{n=1}^{(N-1)/2}c(n)\sin(\omega n)$$

情况 4：$h(n)$为奇对称，N 为偶数

$$H(\omega)=\sum_{n=1}^{N/2}d(n)\sin[(n-\frac{1}{2})\omega]$$

应用简单的三角恒等式，可以将上述公式写成如下形式

情况 1：$$H(\omega)=\sum_{n=0}^{(N-1)/2}\tilde{a}(n)\cos(n\omega)$$

情况 2：$$H(\omega)=\cos(\frac{\omega}{2})\sum_{n=0}^{N/2-1}\tilde{b}(n)\cos(n\omega)$$

情况 3：$$H(\omega)=\sin(\omega)\sum_{n=0}^{(N-3)/2}\tilde{c}(n)\cos(n\omega)$$

情况 4：$$H(\omega)=\sin(\frac{\omega}{2})\sum_{n=0}^{N/2-1}\tilde{d}(n)\cos(n\omega)$$

由此可见，$H(\omega)$可以写成两项乘积的统一形式，即

$$H(\omega)=Q(\omega)P(\omega)$$

其中 $Q(\omega)$是 ω 的固定函数，$P(\omega)$是余弦函数的线性组合。

2. FIR 数字滤波器的设计转换为切比雪夫逼近问题

设理想幅频响应为 $H_d(\omega)$，逼近误差正加权函数为 $W(\omega)$，于是用 $H(\omega)$逼近 $H_d(\omega)$的误差函数 $E(\omega)$定义为

$$E(\omega) = W(\omega)[H_d(\omega) - H(\omega)]$$

将 $H(\omega)=Q(\omega)P(\omega)$ 代入上式，并令 $\hat{H}_d(\omega)=H_d(\omega)/Q(\omega)$，$\hat{W}(\omega)=W(\omega)Q(\omega)$，则有

$$E(\omega) = \hat{W}(\omega)[\hat{H}_d(\omega) - P(\omega)]$$

这样，FIR 数字滤波器的设计可归结为切比雪夫逼近问题，即寻找 $P(\omega)$的一组系数$\tilde{a}(n)$、$\tilde{b}(n)$、$\tilde{c}(n)$或$\tilde{d}(n)$，使在各逼近的频带上，$E(\omega)$的最大绝对值达到最小。若用 $\| E(\omega) \|$ 表示这个最小值，那么切比雪夫逼近问题可表示为

$$\| E(\omega) \| = \min [\max_{\omega \in A} |E(\omega)|]$$

其中 A 表示各频带的并集。

3. 求解切比雪夫逼近问题

依据交错定理，应用 Remez 多重交换算法，可以有效地求解上述切比雪夫逼近问题。

三、使用说明

1. 子函数语句

void fircad(n,type,nbands,lgrid,edge,fx,wtx,h)

2. 形参说明

n ——整型变量。滤波器的阶数。

type——整型变量。滤波器的类型。取值为 1、2 和 3，分别表示多带滤波器、微分器和希尔伯特变换器。

nbands——整型变量。滤波器的频带数，包括通带和阻带。

lgrid——整型变量。分格密度。该值为 0 时，程序自动置为 16。

edge ——双精度实型一维数组，长度为 2 * nbands。滤波器各频带的下边界和上边界值。程序中把频率值归一化到 0.5。

fx ——双精度实型一维数组，长度为 nbands。滤波器各频带的幅度值。对于微分器，则表示斜率值。

wtx ——双精度实型一维数组，长度为 nbands。滤波器各频带的加权值。对于微分器，加权函数反比于频率值。

h ——双精度实型一维数组，长度为(n+1)。存放 FIR 滤波器的系数。

四、子函数程序(文件名:fircad.c)

```
#include "math.h"
#include "stdio.h"
void fircad(n,type,nbands,lgrid,edge,fx,wtx,h)
int n,type,lgrid,nbands;
double h[],fx[],edge[],wtx[];
{ int j,k,l,ix,nz,neg,nm1,kup,nfcns,ngrid;
  int iext[67],band,nodd,nzmj,nf2j,nf3j;
  double pi,xt,dev,pi2,delf,fup,temp,change;
```

```
double ad[67],x[67],y[67],alpha[67];
double grid[1046],des[1046],wt[1046],deviat[11];
double eff(),wate();
void remez();
pi = 4.0 * atan(1.0);
pi2 = 2.0 * pi;
if (nbands <= 0) nbands = 1;
if (lgrid <= 0) lgrid=16;
neg = 1;
if (type == 1) neg = 0;
nodd = n/2;
nodd = n - 2 * nodd;
nfcns = n/2;
if( (nodd == 1) && (neg == 0) ) nfcns++;
grid[1] = edge[1];
delf = lgrid * nfcns;
delf = 0.5/delf;
if (neg != 0)
  { if (edge[1] < delf) grid[1] = delf; }
j = 1; l = 1;
band = 1;
do
  { fup = edge[l+1];
    do
      { temp = grid[j];
        des[j] = eff(temp,fx,band,type);
        wt[j] = wate(temp,fx,wtx,band,type);
        j++;
        grid[j] = temp + delf;
      } while (grid[j] <= fup);
    grid[j-1] = fup;
    des[j-1] = eff(fup,fx,band,type);
    wt[j-1] = wate(fup,fx,wtx,band,type);
    band++;
    l += 2;
    if (band <= nbands) grid[j] = edge[l];
  } while (band <= nbands);
ngrid = j - 1;
```

```
if ( (neg == nodd) && ( grid[ngrid] > (0.5-delf) ) ) ngrid--;
if (neg <= 0)
  { if (nodd != 1)
      { for (j=1;j<=ngrid;j++)
          { change = cos(pi * grid[j]);
            des[j] /= change;
            wt[j] *= change;
          }
      }
  }
else
  { if (nodd != 1)
      { for (j=1;j<=ngrid;j++)
          { change = sin(pi * grid[j]);
            des[j] /= change;
            wt[j] *= change;
          }
      }
    else
      { for (j=1;j<=ngrid;j++)
          { change = sin(pi2 * grid[j]);
            des[j] /= change;
            wt[j] *= change;
          }
      }
  }
temp = (ngrid -1)/((double)nfcns);
for (j=1;j<=nfcns;j++)
  { xt = j - 1;
    iext[j] = xt * temp + 1.0;
  }
iext[nfcns+1] = ngrid;
nm1 = nfcns - 1;
nz = nfcns + 1;
remez(nfcns,ngrid,&dev,ad,x,y,alpha,grid,des,wt,iext);
if (neg <= 0)
  { if (nodd == 0)
      { h[1] = 0.25 * alpha[nfcns];
```

```
            for (j=2;j<=nm1;j++)
              { nzmj = nz - j;
                nf2j = nfcns + 2 - j;
                h[j] = 0.25 * (alpha[nzmj]+alpha[nf2j]);
              }
            h[nfcns] = 0.5 * alpha[1]+0.25 * alpha[2];
          }
        else
          { for (j=1;j<=nm1;j++)
              { nzmj = nz - j;
                h[j] = 0.5 * alpha[nzmj];
              }
            h[nfcns]=alpha[1];
          }
      }
    else
      { if (nodd! =0)
          { h[1] = 0.25 * alpha[nfcns];
            h[2] = 0.25 * alpha[nm1];
            for (j=3;j<=nm1;j++)
              { nzmj = nz - j;
                nf3j = nfcns + 3 - j;
                h[j] = 0.25 * (alpha[nzmj] - alpha[nf3j]);
              }
            h[nfcns] = 0.5 * alpha[1] - 0.25 * alpha[3];
            h[nz] = 0.0;
          }
        else
          { h[1] = 0.25 * alpha[nfcns];
            for (j=2;j<=nm1;j++)
              { nzmj = nz - j;
                nf2j = nfcns + 2 - j;
                h[j] = 0.25 * (alpha[nzmj] - alpha[nf2j]);
              }
            h[nfcns] = 0.5 * alpha[1] - 0.25 * alpha[2];
          }
      }
    printf("\n\n");
```

```
for (j=1;j<=70;j++)
  { printf(" * "); }
printf("\n                    Finite Impulse Response (FIR)");
printf("\n                 Linear Phase Digital Filter Design");
printf("\n                      Remez Exchange Algorithm\n");
if (type == 1) printf("\n                         bandpass filter\n");
if (type == 2) printf("\n                        differentiator\n");
if (type == 3) printf("\n                      hilhert transformer\n");
printf("\n                   filter order =    %d\n",n-1);
printf("\n                   * * * * * impulse response * * * * *\n");
for (j=1;j<=nfcns;j+-)
  { k = n + 1 - j;
    if ( neg == 0)
      {printf("           h(%2d) = %14.8lf = h(%3d)\n",j-1,h[j],k-1);
        h[k] = h[j];
      }
    if (neg == 1)
      {printf("           h(%2d)=%14.8lf=-h(%3d)\n",j-1,h[j],k-1);
        h[k] = -h[j];
      }
  }
if ( (neg == 1 ) && ( nodd == 1) )
  { printf("              h(%2d)=%14.8lf\n",nz-1,0.0);}
for (j=0;j<n;j++)
  { h[j] = h[j+1]; }
h[n] = 0.0;
for (k=1;k<=nbands;k+=4)
  { kup = k + 3;
    if (kup > nbands) kup = nbands;
    printf("\n\n                          ");
    for (j=k;j<=kup;j++)
      { printf("%s%3d          ","Band",j); }
    printf("\n lower band edge");
    for (j=k;j<=kup;j++)
      { printf( "%14.7lf ",edge[2*j-1] ); }
    printf("\n upper hand edge");
    for (j=k;j<=kup;j++)
      { printf( "%14.7lf ",edge[2*j] ); }
```

```
    if (type != 2)
      { printf("\n desired value   ");
        for (j=k;j<=kup;j++)
          { printf( "%14.7lf ",fx[j]); }
          }
    if (type == 2)
      { printf("\n desired slope   ");
        for (j=k;j<=kup;j++)
          { printf( "%14.7lf ",fx[j]); }
      }
    printf("\n weight            ");
    for (j=k;j<=kup;j++)
      { printf( "%14.7lf ",wtx[j]); }
    for (j=k;j<=kup;j++)
      { deviat[j]=dev/wtx[j]; }
    printf("\n deviation         ");
    for (j=k;j<=kup;j++)
      { printf( "%14.7lf ",deviat[j]); }
    if (type != 1) continue;
    for (j=k;j<=kup;j++)
      { deviat[j] = 20.0 * log10(deviat[j] + fx[j]); }
    printf("\n deviation in db");
    for (j=k;j<=kup;j++)
      { printf( "%14.7lf ",deviat[j] ); }
  }
for (j=1;j<=nz;j++)
  { ix = iext[j];
    grid[j] = grid[ix];
  }
printf("\n\n extremal frequencies--maxima of the error curve\n");
for (j=1;j<=nz;j++)
  { printf(" %12.7lf",grid[j]);
    if (j%5 == 0) printf("\n");
  }
printf("\n\n\n");
for (j=1;j<=70;j++)
  { printf(" * "); }
printf("\n");
```

```
}

static double eff(freq,fx,band,type)
double freq,fx[ ];
int band,type;
{ double effv;
  effv = ( type ! = 2 ) ? fx[band] : fx[band]*freq;
  return(effv);
}

static double wate(freq,fx,wtx,band,type)
int band,type;
double freq, fx[ ],wtx[ ];
{ double wate-v;
  if (type ! = 2)
     { wate-v = wtx[band]; }
  else
     { if (fx[band] >= 0.0001)
          { wate-v = wtx[band]/freq; }
       else
          { wate-v = wtx[band]; }
     }
  return(wate-v);
}

static void remez(nfcns,ngrid,pdev,ad,x,y,alpha,grid,des,wt,iext)
int nfcns,ngrid,iext[];
double *pdev;
double ad[],x[],y[],alpha[],grid[],des[],wt[];
{ int j,k,l,k1,nu,nzz,jxt,jet,knz,klow,nut,niter,itrmax,jchnge;
  int nz,kup,nut1,luck,nzzmj,kn,nm1,nzmj,kkk,jm1,jp1,nf1j;
  double dk,dak,pi2,dnum,dden,dtemp,ynz,y1,err;
  double aa,bb,cn,ft,xt,xt1,xe,fsh,gtemp,delf;
  double comp=0.0,devl=-1.0;
  double a[67],p[67],q[67];
  double d(),gee();
  void ouch();
  pi2 = 8.0*atan(1.0);
```

```
itrmax = 25;
nz = nfcns + 1;
nzz = nfcns + 2;
niter = 0;
while ( 1 )
  { iext[nzz] = ngrid + 1;
    niter++;
    if (niter > itrmax) goto l9;
    for (j=1;j<=nz;j++)
      { jxt = iext[j];
        dtemp = grid[jxt];
        dtemp = cos(dtemp * pi2);
        x[j] = dtemp;
      }
    jet = (nfcns-1)/15 + 1;
    for (j=1;j<=nz;j++)
      { ad[j] = d(j,nz,jet,x); }
    dnum = 0.0;
    dden = 0.0;
    k = 1;
    for (j=1;j<=nz;j++)
      { l = iext[j];
        dtemp = ad[j] * des[l];
        dnum += dtemp;
        dtemp = ((double)k) * ad[j]/wt[l];
        dden += dtemp;
        k = 0 - k;
      }
    *pdev = dnum/dden;
    nu = 1;
    if ( *pdev > 0.0) nu = -1;
    *pdev *= (-(double)nu);
    k = nu;
    for (j=1;j<=nz;j++)
      { l = iext[j];
        dtemp = ( (double)k) * ( *pdev)/wt[l];
        y[j] = des[l] + dtemp;
        k = -k;
```

```
    }
  if ( *pdev <= devl)
    { ouch(niter);
      goto l9;
    }
  devl = *pdev;
  jchnge = 0;
  k1 = iext[1];
  knz = iext[nz];
  klow = 0;
  nut = -nu;
  j = 1;
  while ( 1 )
    { if (j == nzz) ynz = comp;
      if (j >= nzz) break;
      kup = iext[j+1];
      l = iext[j] + 1;
      nut = -nut;
      if (j == 2) y1 = comp;
      comp = *pdev;
      if (l >= kup) goto l3;
      err = gee(l,nz,ad,x,y,grid);
      err = (err - des[l]) * wt[l];
      dtemp = ( (double)nut ) * err-comp;
      if (dtemp <= 0.0) goto l3;
      comp = (double)nut * err;
l1:
      while ( 1 )
        { l++;
          if (l >= kup) break;
          err = gee(l,nz,ad,x,y,grid);
          err = (err-des[l]) * wt[l];
          dtemp = (double)nut * err - comp;
          if (dtemp <= 0.0) break;
          comp = (double)nut * err;
        }
l2:
      iext[j] = l - 1;
```

```
        j++;
        klow = l - 1;
        jchnge++;
        continue;
13:
        l--;
        do
          { l--;
            if (l <= klow) goto 16;
            err = gee(l,nz,ad,x,y,grid);
            err = (err-des[l]) * wt[l];
            dtemp = (double)nut * err - comp;
            if (dtemp >= 0.0) goto 14;
          } while (jchnge <= 0);
        klow = iext[j];
        j++;
        continue;
14:
        comp = (double)nut * err;
15:
        while ( 1 )
          { l--;
            if (l <= klow) break;
            err = gee(l,nz,ad,x,y,grid);
            err = (err-des[l]) * wt[l];
            dtemp = (double)nut * err-comp;
            if (dtemp <= 0.0) break;
            comp = (double)nut * err;
          }
        klow = iext[j];
        iext[j] = l + 1;
        j++;
        jchnge++;
        continue;
16:
        l = iext[j] + 1;
        if (jchnge > 0) goto 12;
17:
```

```
      l++;
      if (l >= kup)
        { klow = iext[j];
          j++;
          continue;
        }
      err = gee(l,nz,ad,x,y,grid);
      err = (err-des[l]) * wt[l];
      dtemp = (double)nut * err - comp;
      if (dtemp <= 0.0) goto l7;
      comp = (double)nut * err;
      goto l1;
    }
  if (j <= nzz)
    { if (k1 > iext[1]) k1 = iext[1];
      if (knz < iext[nz]) knz = iext[nz];
      nut1 = nut;
      nut = -nu;
      l = 0;
      kup = k1;
      comp = ynz * (1.00001);
      luck = 1;
      do
        { l++;
          if (l >= kup)
            { luck = 6;
              goto l8;
            }
          err = gee(l,nz,ad,x,y,grid);
          err = (err - des[l]) * wt[l];
          dtemp = (double)nut * err - comp;
        } while (dtemp <= 0.0) ;
      comp = (double)nut * err;
      j = nzz;
      goto l1;
    }
  if (luck > 9)
    { kn = iext[nzz];
```

```
        for (j=1;j<=nfcns;j++)
          { iext[j] = iext[j+1]; }
        iext[nz] = kn;
        continue;
      }
    if (comp > y1) y1 = comp;
    k1 = iext[nzz];
18:
    l = ngrid + 1;
    klow = knz;
    nut = -nut1;
    comp = y1 * (1.00001);
    do
      { l--;
        if (l <= klow) break;
        err = gee(l,nz,ad,x,y,grid);
        err = (err-des[l]) * wt[l];
        dtemp = (double)nut * err - comp;
      } while (dtemp <= 0.0);
    if (l > klow)
      { j = nzz;
        comp = (double)nut * err;
        luck += 10;
        goto 15;
      }
    if (luck == 6)
      { if (jchnge > 0)
          { continue; }
        else
          { break; }
      }
    for (j=1;j<=nfcns;j++)
      { nzzmj = nzz - j;
        nzmj = nz - j;
        iext[nzzmj] = iext[nzmj];
      }
    iext[1] = k1;
  }
```

```
l9:
  nm1 = nfcns - 1;
  fsh = 1.0e-06;
  gtemp = grid[1];
  x[nzz] = -2.0;
  cn = 2 * nfcns - 1;
  delf = 1.0/cn;
  l = 1;
  kkk = 0;
  if ( (grid[1] < 0.01) && (grid[ngrid] > 0.49) ) kkk = 1;
  if (nfcns <= 3) kkk = 1;
  if (kkk != 1)
    { dtemp = cos(pi2 * grid[1]);
      dnum = cos(pi2 * grid[ngrid]);
      aa = 2.0/(dtemp-dnum);
      bb = -(dtemp+dnum)/(dtemp-dnum);
    }
  for (j=1;j<=nfcns;j++)
    { ft = j - 1;
      ft *= delf;
      xt = cos(pi2 * ft);
      if (kkk != 1)
        { xt = (xt-bb)/aa;
          xt1 = sqrt(1.0-xt * xt);
          ft = atan2(xt1,xt)/pi2;
        }
      while ( 1 )
        { xe = x[l];
          if (xt > xe) break;
          if ((xe-xt) < fsh)
            { a[j] = y[l];
              goto l10;
            }
          l++;
        }
      if ((xt-xe) < fsh)
        { a[j] = y[l];
          if (l > 1) l--;
```

```
            continue;
          }
        grid[1] = ft;
        a[j] = gee(1,nz,ad,x,y,grid);
l10:
        if (l > 1) l--;
      }
    grid[1] = gtemp;
    dden = pi2/cn;
    for (j=1;j<=nfcns;j++)
      { dtemp = 0.0;
        dnum = j - 1;
        dnum *= dden;
        if (nm1 >= 1)
          { for (k=1;k<=nm1;k++)
              { dak = a[k+1];
                dk = k;
                dtemp = dtemp + dak * cos(dnum * dk);
              }
          }
        dtemp = 2.0 * dtemp + a[1];
        alpha[j] = dtemp;
      }
    for (j=2;j<=nfcns;j++)
      { alpha[j] *= 2.0/cn; }
    alpha[1] /= cn;
    if (kkk != 1)
      { p[1] = 2.0 * alpha[nfcns] * bb + alpha[nm1];
        p[2] = 2.0 * aa * alpha[nfcns];
        q[1] = alpha[nfcns-2] - alpha[nfcns];
        for (j=2;j<=nm1;j++)
          { if (j >= nm1)
              { aa *= 0.5;
                bb *= 0.5;
              }
            p[j+1] = 0.0;
            for (k=1;k<=j;k++)
              { a[k] = p[k];
```

```
                p[k] = 2.0 * bb * a[k];
              }
            p[2] += a[1] * 2.0 * aa;
            jm1 = j - 1;
            for (k=1;k<=jm1;k++)
              { p[k] += q[k] + aa * a[k+1]; }
            jp1 = j + 1;
            for (k=3;k<=jp1;k++)
              { p[k] += aa * a[k-1]; }
            if (j == nm1) continue;
            for (k=1;k<=j;k++)
              { q[k] = -a[k]; }
            nf1j = nfcns-1-j;
            q[1] += alpha[nf1j];
          }
        for (j=1;j<=nfcns;j++)
          { alpha[j] = p[j]; }
      }
    if (nfcns <= 3)
      { alpha[nfcns+1] = 0.0;
        alpha[nfcns+2] = 0.0;
      }
  }

static double d(k,nz,m,x)
int k,m,nz;
double x[];
{ int j,l;
  double q,dv;
  dv = 1.0;
  q = x[k];
  for (l=1;l<=m;l++)
    { for(j=l;j<=nz;j+=m)
        { if (j == k) continue;
          dv = 2.0 * dv * (q-x[j]);
        }
    }
  dv = 1.0/dv;
```

```
  return(dv);
}

static double gee(k,nz,ad,x,y,grid)
int k,nz;
double ad[],x[],y[],grid[];
{ int j;
  double c,d,p,xf,pi2,geev;
  d = p =0.0;
  pi2 = 8.0*atan(1.0);
  xf = grid[k];
  xf = cos(pi2*xf);
  for (j=1;j<=nz;j++)
    { c = xf - x[j];
      c = ad[j]/c;
      d += c;
      p += c*y[j];
    }
  geev = p/d;
  return(geev);
}

static void ouch(niter)
int niter;
{ printf("\n ********* failure to converge *********\n\n");
  printf("\nprobable cause is machine rounding error\n\n");
  printf("\nnumber of iterations =%d",niter);
  printf("\nif the number of iterations exceeds 3");
  printf("\ntbe design may be correct, but should be verified");
  printf(" with an FFT.");
}
```

五、例题

下面给出主函数程序，它调用 fircad()子函数。通过人机对话输人参数后，它可以设计线性相位 FIR 数字滤波器、微分器和希尔伯特变换器。此处要调用计算滤波器频率响应的函数 gain()，参看第二篇 § 4.1 节。

主函数程序(文件名:fircad.m):

```
#include "stdio.h"
```

```
#include "fircad.c"
#include "gain.c"
main()
{ int i,j,jb,n,type,lgrid,nbands;
  double fx[30],edge[30],wtx[30];
  double freq,h[150],c[150],x[300],y[300];
  char fname[40];
  FILE *fp;
  c[1]=0.0;
  printf("input the filter order:\n");
  scanf("%d",&n);
  printf("input the filter type:\n");
  printf("    1 = multiple passband/stopband filter\n");
  printf("    2 = differentiator\n");
  printf("    3 = hilbert transform filter\n");
  scanf("%d",&type);
  printf("input the number of bands:\n");
  scanf("%d",&nbands);
  printf("input the number of grids ,if 0 it is 16:\n");
  scanf("%d",&lgrid);
  jb = 2*nbands;
  printf("input the band edges:\n");
  for (j=1;j<=jb;j++)
    { scanf("%lf",&edge[j]); }
  printf("input the desired response amplitude of each bands:\n");
  for (j=1;j<=nbands;j++)
    { scanf("%lf",&fx[j]); }
  printf("input the weight of each bands:\n");
  for (j=1;j<=nbands;j++)
    { scanf("%lf",&wtx[j]); }
  fircad(n+1,type,nbands,lgrid,edge,fx,wtx,h);
  printf("\ninput file name of frequency response\n");
  scanf("%s",fname);
  if ( (fp=fopen(fname,"w")) == NULL )
    { printf("cannot open this file\n");
      exit(1);
    }
  gain(h,c,n,1,x,y,300,2);
```

```
    for (i=0;i<300;i++)
      { freq = 0.5*i/299;
        fprintf(fp,"%lf        %lf \n",freq,x[i]);
      }
    fclose(fp);
}
```

例1：设计一个32阶的带通数字滤波器，其通带为0.2～0.35，阻带为0.0～0.1和0.425～0.5，通带加权值为1，阻带加权值为10，分格密度为32。这时输入

```
32   1   3   32
0.0   0.1   0.2   0.35   0.425   0.5
0.0   1.0   0.0
10   1   10
```

运行结果：

滤波器的系数为

```
h( 0) =   -0.00269911   = h(32)
h( 1) =   -0.00451944   = h(31)
h( 2) =    0.00818165   = h(30)
h( 3) =   -0.00097323   = h(29)
h( 4) =   -0.00035092   = h(28)
h( 5) =    0.01708742   = h(27)
h( 6) =   -0.01740830   = h(26)
h( 7) =   -0.00261714   = h(25)
h( 8) =   -0.01016807   = h(24)
h( 9) =   -0.03730066   = h(23)
h(10) =    0.06346405   = h(22)
h(11) =    0.01856136   = h(21)
h(12) =    0.02785976   = h(20)
h(13) =    0.04967215   = h(19)
h(14) =   -0.30023050   = h(18)
h(15) =   -0.04071342   = h(17)
h(16) =    0.46270287   = h(16)
```

带通数字滤波器的幅频响应如图2－6－6所示。

例2：设计一个31阶的宽带微分器，其斜率为1。这时输入

```
31   2   1   0
0.0   0.5
1.0
1.0
```

运行结果：

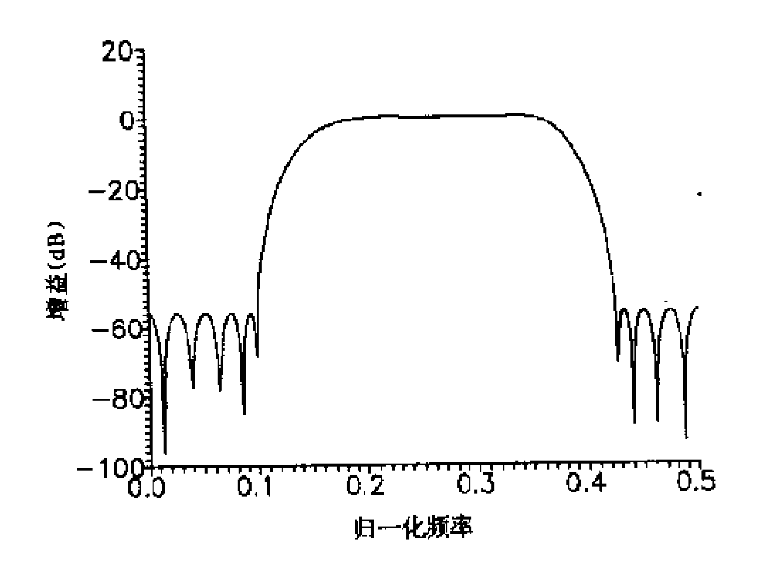


图 2—6—6 带通 FIR 数字滤波器的幅频响应

数字微分器的系数为

h(0) = −0.00062713 = h(31)
h(1) = 0.00085633 = h(30)
h(2) = −0.00042419 = h(29)
h(3) = 0.00039902 = h(28)
h(4) = −0.00043437 = h(27)
h(5) = 0.00049969 = h(26)
h(6) = −0.00059635 = h(25)
h(7) = 0.00073277 = h(24)
h(8) = −0.00093003 = h(23)
h(9) = 0.00122700 = h(22)
h(10) = −0.00170128 = h(21)
h(11) = 0.00252723 = h(20)
h(12) = −0.00416012 = h(19)
h(13) = 0.00812946 = h(18)
h(14) = −0.02253910 = h(17)
h(15) = 0.20266535 = h(16)

数字微分器的幅频响应如图 2—6—7 所示。

例 3：设计一个 19 阶的希尔伯特变换器，其通带下边界频率为 0.05，通带上边界频率为 0.5。这时输入

```
19   3   1   16
0.05   0.5
1.0
1.0
```

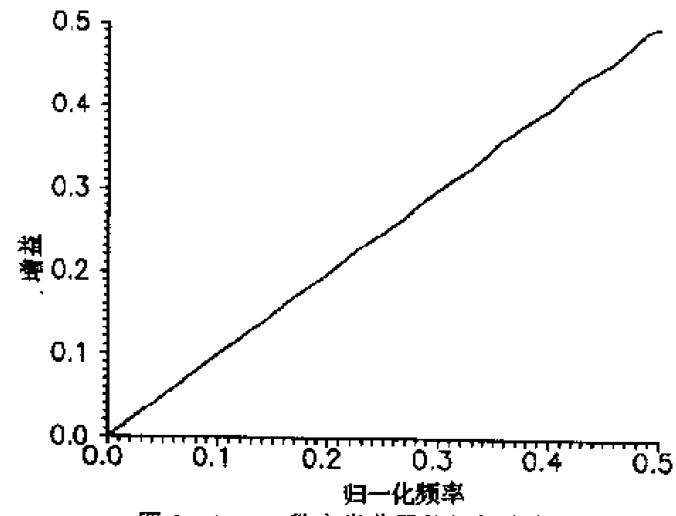


图 2—6—7 数字微分器的幅频响应

运行结果：

滤波器的系数为

h(0) = 0.01602620 = h(19)

h(1) = 0.01417329 = h(18)

h(2) = 0.02045244 = h(17)

h(3) = 0.02873689 = h(16)

h(4) = 0.03985258 = h(15)

h(5) = 0.05533330 = h(14)

h(6) = 0.07854276 = h(13)

h(7) = 0.11823756 = h(12)

h(8) = 0.20664125 = h(11)

h(9) = 0.63475618 = h(10)

希尔伯特变换器的幅频响应如图 2—6—8 所示。

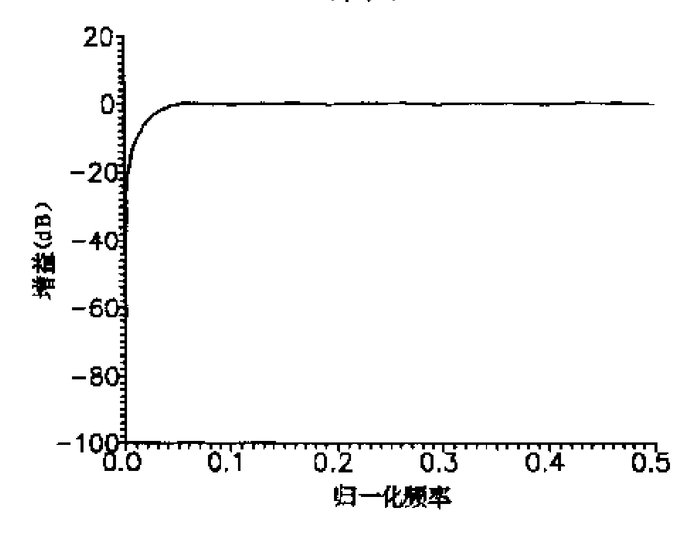


图 2—6—8 希尔伯特变换器的幅频响应

第三篇　随机数字信号处理

第一章　经典谱估计

§1.1　功率谱估计的周期图方法

一、功　能

用 Welch 的平均周期图方法来计算信号的自功率谱和相关函数。

二、方法简介

设有限长序列 $x(n)(n=0,1,\cdots,N-1)$ 的功率谱为 $S_{xx}(k)$。用平均周期图方法进行功率谱估计的计算过程如下：

1. 把 $x(n)$ 分成长度为 M 的 K 段，相邻段重叠 $\frac{M}{2}$ 个样本，同时对每段数据加窗处理，这样第 i 段数据为

$$x_i(n) = x(iM/2 + n)\, w(n) \quad , \quad 0 \leqslant n \leqslant M-1, \quad 0 \leqslant i \leqslant K-1$$

其中 $w(n)$ 为窗函数(如矩形窗、海明窗等)。

2. 用 FFT 算法计算序列 $x_i(n)$ 的 L 点离散傅立叶变换 $X_i(k)$

$$X_i(k) = \sum_{n=0}^{L-1} x_i(n) e^{-j\frac{2\pi}{L}kn} \quad , \quad 0 \leqslant k \leqslant L-1, \quad 0 \leqslant i \leqslant K-1$$

如果 $M<L$，序列 $x_i(n)$ 要补 $L-M$ 个零。

3. 计算周期图

$$S_i(k) = |X_i(k)|^2 \quad , \quad 0 \leqslant k \leqslant L-1, \quad 0 \leqslant i \leqslant K-1$$

4. 计算 K 段周期图的平均值，从而得到功率谱的估值为

$$S_{xx}(k) = \frac{1}{KU}\sum_{i=0}^{K-1} S_i(k) \quad , \quad 0 \leqslant k \leqslant L-1$$

其中

$$U = \sum_{n=0}^{M-1} w^2(n)$$

三、使用说明

1. 子函数语句

```
void pmpse(x,n,m,nfft,win,fs,r,freq,sxx,sdh)
```

2. 形参说明

x ——双精度实型一维数组，长度为n。输入信号 $x(i)$。

n ——整型变量。输入信号的长度。

m ——整型变量。分段的长度。

nfft ——整型变量。估计功率谱所用 FFT 的长度。它必须是 2 的整数次幂且 nfft≥m。

win ——整型变量。窗函数的类型。win=1，表示矩形窗；win=2，表示海明(Hamming)窗。

f_s——双精度实型变量。采样频率(以赫兹为单位)。

r ——双精度实型一维数组，长度为$(\frac{nfft}{2}+1)$。用于存放相关函数 $r(i)$的值。

freq ——双精度实型一维数组，长度为$(\frac{nfft}{2}+1)$。用于存放与功率谱相对应的频率值。

sxx ——双精度实型一维数组，长度为$(\frac{nfft}{2}+1)$。用于存放功率谱的值。

sdb ——整型变量。用于表示功率谱的类型。sdb=0，表示线性谱；sdb=1，表示以 dB 为单位的对数谱。

四、子函数程序(文件名：pmpse.c)

```
#include "math.h"
#include "stdlib.h"
#include "fft.c"
void pmpse(x,n,m,nfft,win,fs,r,freq,sxx,sdb)
int nfft,n,win,m,sdb;
double fs,x[],r[],freq[],sxx[];
{ int i,j,k,s;
  int m2,nrd,kmax,ns1,nsectp;
  int nfft21,NumOfSections,NumUsed;
  double u,fl,xsum,norm,twopi,rexmn,imxmn,xmean;
  double  *xa, *xreal, *ximag, *window;
  xa = malloc(nfft * sizeof(double));
  xreal = malloc(nfft * sizeof(double));
  ximag = malloc(nfft * sizeof(double));
  window = malloc(m * sizeof(double));
  nfft21 = nfft/2 + 1;
  NumOfSections = (n - m/2)/( m/2 );
  NumUsed = NumOfSections * (m/2) + m/2;
  s = 0;
  xsum = 0.0;
```

```
ns1 = NumOfSections + 1;
m2 = m/2;
for ( k=0; k<ns1; k++)
  { for (i=0;i<m2;i++)
      { xa[i] = x[s+i]; }
    for (i=0; i<m2; i++)
      { xsum = xsum + xa[i]; }
    s += m2;
  }
xmean = xsum/NumUsed;
rexmn = xmean;
imxmn = xmean;
u = (double)m ;
if ( win == 2 )
  { u = 0.0;
    twopi = 8.0 * atan(1.0);
    fl = m - 1.0;
    for (i=0;i<m;i++)
      { window[i] = 0.54 - 0.46 * cos(twopi * i/fl);
        u += window[i] * window[i];
      }
  }
s = 0;
for (i=0; i<nfft21; i++)
  { sxx[i] = 0.0; }
m2 = m/2;
for (i=0;i<m2;i++)
  { xa[i+m2] = x[s+i]; }
s += m2;
kmax = (NumOfSections + 1)/2;
nsectp = (NumOfSections + 1)/2;
nrd = m;
for (k=0; k<kmax; k++)
  { for (i=0; i<m2; i++)
      { j = m2 + i;
        xreal[i] = xa[j];
        ximag[i] = 0.0;
      }
```

```
if ( (k == (kmax-1)) && (nsectp != NumOfSections))
  { for (i=m2; i<nrd; i++)
      { xa[i] = 0.0; }
    nrd = m/2;
  }
for (i=0;i<nrd;i++)
  { xa[i] = x[s+i]; }
for ( i=0; i<m2; i++)
  { j = m2 + i ;
    xreal[j] = xa[i] - rexmn;
    ximag[j] = xa[j] - imxmn;
    xreal[i] = xreal[i] -rexmn;
    ximag[i] = xa[i] - imxmn;
  }
if ( (k == (kmax-1) ) && (nsectp != NumOfSections) )
  { for (i=0; i<m; i++)
      { ximag[i] = 0.0; }
  }
s = s + nrd;
if (win == 2)
  { for (i=0;i<m;i++)
      { xreal[i] *= window[i];
        ximag[i] *= window[i];
      }
  }
if (m != nfft)
  { for (i=m;i<nfft;i++)
      { xreal[i] = 0.0;
        ximag[i] = 0.0;
      }
  }
fft(xreal,ximag,nfft,1);
for (i=1; i<nfft21; i++)
  { j = nfft - i;
    sxx[i] += xreal[i] * xreal[i] + ximag[i] * ximag[i];
    sxx[i] += xreal[j] * xreal[j] + ximag[j] * ximag[j];
  }
sxx[0] += xreal[0] * xreal[0] * 2.0;
```

```
            sxx[0] += ximag[0] * ximag[0] * 2.0;
         }
       norm = 2.0 * u * NumOfSections;
       for ( i=0; i<nfft21; i++)
         { sxx[i] = sxx[i]/norm;
           xreal[i] = sxx[i];
           ximag[i] = 0.0;
           j = nfft - i;
           xreal[j] = xreal[i] ;
           ximag[j] = ximag[i] ;
         }
       fft(xreal,ximag,nfft,-1);
       for (i=0; i<nfft21; i++)
         { r[i] = xreal[i]; }
       for (i=0; i<nfft21; i++)
         { freq[i] = i * fs/(double)nfft;
           if (sdb == 1)
             { if (sxx[i] == 0.0) sxx[i] = 1.0e-15;
               sxx[i] = 20.0 * log10(sxx[i]);
             }
         }
       free(xa);
       free(xreal);
       free(ximag);
       free(window);
     }
```

五、例　题

下面给出主函数程序，它调用子函数 pmpse()。通过人机对话输入参数后，它能计算功率谱和相关函数，同时将功率谱数据存入文件 pmpse. dat 中。

例：信号 $x(i)$是频率为 f 赫兹的余弦函数

$$x(i) = \cos(2\pi i f/f_s)$$

选择参数为：信号频率 f=1000Hz，采样频率 f_s=10000Hz，数据长度 n=256，每段长度 m=64，FFT 长度 nfft=128，海明窗函数，输出功率谱用 dB 表示。

主函数程序(文件名:pmpse. m)：

```
#include "stdio.h"
#include "math.h"
#include "pmpse.c"
```

```
main()
{ int i,m,n,win,sdb,nfft,nfft21;
  double fs,pi,freq[100],sxx[100],r[100],x[512];
  FILE *fp;
  pi = 4.0 * atan(1.0);
  printf("\ninput the sample number n\n");
  scanf("%d",&n);
  printf("sampling frequency fs\n");
  scanf("%lf",&fs);
  printf("input section size m\n");
  scanf("%d",&m);
  printf("input the window type: 1 for rectangular; 2 for hamming\n");
  scanf("%d",&win);
  printf("input the size of FFT for spectral estimate\n");
  scanf("%d",&nfft);
  printf("input type of spectrum: 0 for linear, 1 for log spec in dB\n");
  scanf("%d",&sdb);
  for (i=0; i<n; i++)
    { x[i] = cos(2*pi*i*1000/fs); }
  pmpse(x,n,m,nfft,win,fs,r,freq,sxx,sdb);
  nfft21 = nfft/2+1;
  printf("\nLog Power Spectrum:\n");
  printf("\n        FREQ           dB       FREQ             dB");
  printf("        FREQ           dB\n");
  for (i=0; i<nfft21; i++)
    { printf("%11.1lf %13.4lf",freq[i],sxx[i]);
      if ((i+1)%3 == 0) printf("\n");
    }
  printf("\nCorrelation Function\n");
  for (i=0; i<nfft21; i++)
    { printf("%6d %12.3e ",i,r[i]);
      if ((i+1)%4 == 0) printf("\n");
    }
  fp = fopen("pmpse.dat","w");
  for (i=0; i<nfft21; i++)
    { fprintf(fp,"%11.1lf%13.4lf\n",freq[i],sxx[i]);}
  fclose(fp);
}
```

运行结果：

功率谱为

FREQ	dB	FREQ	dB	FREQ	dB
0.0	−61.4497	78.1	−82.8430	156.2	−61.3074
234.4	−83.4558	312.5	−61.2332	390.6	−86.3288
468.8	−63.3798	546.9	−92.0069	625.0	−67.5814
703.1	−51.5466	781.2	−9.4458	859.4	9.7457
937.5	19.1047	1015.6	21.1584	1093.8	16.3247
1171.9	3.5113	1250.0	−21.7070	1328.1	−89.0637
1406.2	−77.8213	1484.4	−78.8848	1562.5	−64.6268
1640.6	−78.5615	1718.8	−63.1349	1796.9	−80.5160
1875.0	−64.0269	1953.1	−82.6791	2031.2	−65.3888
2109.4	−84.6981	2187.5	−66.7937	2265.6	−86.5087
2343.8	−68.1208	2421.9	−88.1142	2500.0	−69.3368
2578.1	−89.5338	2656.2	−70.4371	2734.4	−90.7892
2812.5	−71.4265	2890.6	−91.8998	2968.8	−72.3131
3046.9	−92.8824	3125.0	−73.1050	3203.1	−93.7508
3281.2	−73.8100	3359.4	−94.5166	3437.5	−74.4349
3515.6	−95.1893	3593.8	−74.9854	3671.9	−95.7768
3750.0	−75.4665	3828.1	−96.2855	3906.2	−75.8824
3984.4	−96.7206	4062.5	−76.2366	4140.6	−97.0865
4218.8	−76.5318	4296.9	−97.3865	4375.0	−76.7704
4453.1	−97.6235	4531.2	−76.9542	4609.4	−97.7995
4687.5	−77.0846	4765.6	−97.9160	4843.8	−77.1624
4921.9	−97.9740	5000.0	−77.1883		

功率谱如图 3−1−1 所示。

相关函数(0～64)为

0	5.00e−01	1	4.04e−01	2	1.54e−01	3	−1.53e−01
4	−3.96e−01	5	−4.83e−01	6	−3.85e−01	7	−1.44e−01
8	1.41e−01	9	3.62e−01	10	4.35e−01	11	3.42e−01
12	1.27e−01	13	−1.22e−01	14	−3.08e−01	15	−3.66e−01
16	−2.83e−01	17	−1.03e−01	18	9.84e−02	19	2.44e−01
20	2.85e−01	21	2.18e−01	22	7.80e−02	23	−7.34e−02
24	−1.79e−01	25	−2.06e−01	26	−1.54e−01	27	−5.43e−02
28	5.04e−02	29	1.21e−01	30	1.36e−01	31	1.00e−01
32	3.46e−02	33	−3.16e−02	34	−7.42e−02	35	−8.21e−02
36	−5.92e−02	37	−1.99e−02	38	1.79e−02	39	4.11e−02
40	4.43e−02	41	3.11e−02	42	1.02e−02	43	−9.04e−03

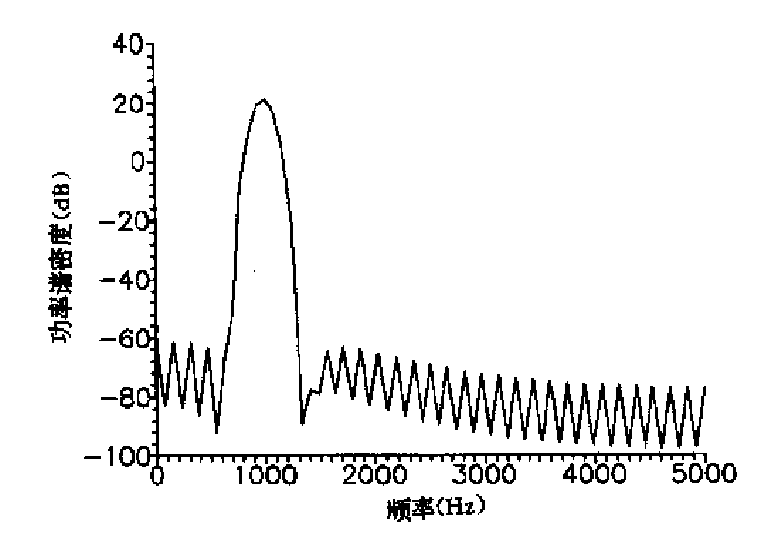


图 3-1-1　信号 $x(i)$的功率谱

44	−2.00e−02	45	−2.10e−02	46	−1.43e−02	47	−4.49e−03
48	3.93e−03	49	8.35e−03	50	8.40e−03	51	5.47e−03
52	1.63e−03	53	−1.41e−03	54	−2.81e−03	55	−2.67e−03
56	−1.64e−03	57	−4.47e−04	58	3.86e−04	59	6.99e−04
60	6.02e−04	61	3.20e−04	62	6.25e−05	63	−5.08e−05
64	−5.55e−17						

§1.2　功率谱估计的相关方法

一、功能

用快速傅立叶变换技术，计算信号的自相关函数或互相关函数，并用 Blackman-Tukey 方法计算其功率谱。

二、方法简介

设两个有限长序列分别为 $x(n)$和 $y(n)$ $(n=0,1,\cdots,N-1)$，它们的相关函数定义为

$$r_{xy}(m)=\frac{1}{N}\sum_{n=0}^{N-1-m}[x(n)-\underline{x}][y(n+m)-\underline{y}]\quad,\quad 0\leqslant m\leqslant L-1$$

其中$\underline{x}$和$\underline{y}$分别是 $x(n)$和 $y(n)$的均值。如果 $x(n)=y(n)$，并且$\underline{x}=\underline{y}=0$，那么上式称为自相关；如果 $x(n)\neq y(n)$，但$\underline{x}=\underline{y}=0$，那么上式称为互相关；如果 $x(n)=y(n)$，但$\underline{x}\neq 0$，$\underline{y}\neq 0$，那么上式称为自协方差；如果 $x(n)\neq y(n)$，并且$\underline{x}\neq 0$，$\underline{y}\neq 0$，那么上式称为互协方差。

一般情况下，$N\gg L$，因此我们常用快速傅立叶变换算法来计算相关函数，详见第二篇§3.4 节。若 FFT 为 M 点，那么可以得到相关函数的$\frac{M}{2}+1$个值。

为了用相关方法计算功率谱，首先对相关函数进行加窗处理

$$\tilde{r}_{xy}(m) = r_{xy}(m)\,w(m) \qquad 0 \leqslant m \leqslant L-1$$

对加窗后的相关函数 $\tilde{r}_{xy}(m)$ 进行快速傅立叶变换，便得到自功率谱或互功率谱 $S_{xy}(k)$

$S_{xy}(k)=\mathrm{DFT}\,[\tilde{r}_{xy}(m)]$

在上述计算中，为减少谱估计的方差，要对数据进行分段处理。

三、使用说明

1. 子函数语句

void cmpse(x,y,n,m,mode,win,fs,lag,nfft,r,freq,sxy,sdb)

2. 形参说明

x ——双精度实型一维数组，长度为n。输入信号 $x(i)$。

y ——双精度实型一维数组，长度为n。输入信号 $y(i)$。

n ——整型变量。输入信号的长度。

m ——整型变量。分段的长度。

mode ——整型变量。相关函数的类型。mode=0，表示自相关；mode=1，表示互相关；mode=2，表示自协方差；mode=3，表示互协方差。

win ——整型变量。窗函数的类型。win=1，表示矩形窗；win=2，表示海明窗。

f_s——双精度实型变量。采样频率(以赫兹为单位)。

lag ——整型变量。估计功率谱所用相关函数的点数。$lag \leqslant \frac{m}{2}+1$。

nfft ——整型变量。估计功率谱所用 FFT 的长度。$nfft \geqslant 2*lag-1$。

r ——双精度实型一维数组，长度为$(\frac{m}{2}+1)$。用于存放相关函数 $r_{xy}(i)$的值。

freq ——双精度实型一维数组，长度为$(\frac{nfft}{2}+1)$。用于存放与功率谱相对应的频率值。

sxy ——双精度实型一维数组，长度为$(\frac{nfft}{2}+1)$。用于存放互功率谱的值。

sdb ——整型变量。用于指示功率谱的单位。sdb=0，表示线性谱；sdb=1，表示以dB 为单位的对数谱。

四、子函数程序(文件名:cmpse.c)

```
#include "math.h"
#include "stdlib.h"
#include "fft.c"
void cmpse(x,y,n,m,mode,win,fs,lag,nfft,r,freq,sxy,sdb)
int m,n,sdb,lag,win,mode,nfft;
double fs,r[],x[],y[],freq[],sxy[];
{ int i,j,k,s;
  int nrd,nrdy,nrdx,shft,mhlf1,nsect1;
  int nrdx1,nlast,nhf1;
```

```
double xmnc,xmns,xic,xis,yic,yis;
double pi,xsum,ysum,xmean,ymean,nsect;
double  *xa, *xc, *xs, *zc, *zs;
xa = malloc(nfft * sizeof(double));
xc = malloc(nfft * sizeof(double));
xs = malloc(nfft * sizeof(double));
zc = malloc(m * sizeof(double));
zs = malloc(m * sizeof(double));
pi = 4.0 * atan(1.0);
shft = m/2;
mhlf1 = shft + 1;
nsect = (n+shft-1.0)/shft;
if (mode >= 2)
  { s = 0;
    nrd = shft;
    xsum = 0.0;
    ysum = 0.0;
    for (k=0;k<nsect;k++)
      { if (k == (nsect-1)) nrd = n - (nsect--1) * nrd;
        for (i=0;i<nrd;i++)
          { xa[i] = x[s+i]; }
        for (i=0;i<nrd;i++)
          { xsum += xa[i]; }
        if (mode ! = 2)
          { for (i=0;i<nrd;i++)
               { xa[i] = y[s+i]; }
            for (i=0;i<nrd;i++)
               { ysum = ysum + xa[i]; }
          }
        s = s + nrd;
      }
    xmean = xsum/n;
    ymean = ysum/n;
    if (mode == 2) ymean = xmean;
    xmnc = xmean;
    xmns = ymean;
  }
s = 0;
```

```
nrdy = m;
nrdx = shft;
for (i=0;i<mhlf1;i++)
  { zc[i] = 0.0;
    zs[i] = 0.0;
  }
for (k=0;k<nsect;k++)
  { nsect1 = nsect - 2;
    if (k >= nsect1)
      { nrdy = n - k * shft;
        if (k == (nsect-1)) nrdx = nrdy;
        if (nrdy != m)
          { for (i=nrdy;i<m;i++)
              { xc[i] = 0.0;
                xs[i] = 0.0;
              }
          }
      }
    for (i=0;i<nrdy;i++)
      { xa[i] = x[s+i]; }
    for (i=0;i<nrdy;i++)
      { xc[i] = xa[i];
        xs[i] = xa[i];
      }
    if ((mode != 0) && (mode != 2))
      { for (i=0;i<nrdy;i++)
          { xa[i] = y[s+i]; }
        for (i=0;i<nrdy;i++)
          { xs[i] = xa[i]; }
      }
    if (mode >= 2)
      { for (i=0;i<nrdy;i++)
          { xc[i] = xc[i] - xmnc;
            xs[i] = xs[i] - xmns;
          }
      }
    nrdx1 = nrdx;
    for (i=nrdx1;i<m;i++)
```

```
        { xc[i] = 0.0; }
      fft(xc,xs,m,1);
      for (i=1;i<shft;i++)
        { j = m - i;
          xic = (xc[i] + xc[j]) * 0.5;
          xis = (xs[i] - xs[j]) * 0.5;
          yic = (xs[i] + xs[j]) * 0.5;
          yis = (xc[j] - xc[i]) * 0.5;
          zc[i] = zc[i] + xic * yic + xis * yis;
          zs[i] = zs[i] + xic * yis - xis * yic;
        }
      zc[0] = zc[0] + xc[0] * xs[0];
      zc[shft] = zc[shft] + xc[shft] * xs[shft];
      s = s + shft;
    }
  for (i=1;i<shft;i++)
    { j = m - i;
      xc[i] = zc[i];
      xs[i] = zs[i];
      xc[j] = zc[i];
      xs[j] = -zs[i];
    }
  xc[0] = zc[0];
  xs[0] = zs[0];
  xc[shft] = zc[shft];
  xs[shft] = zs[shft];
  fft(xc,xs,m,-1);
  for (i=0;i<mhlf1;i++)
    { xa[i] = xc[i]/n; }
  for (i=0;i<mhlf1;i++)
    { r[i] = xa[i]; }
  for (i=1;i<lag;i++)
    { if (win != 1)
        { xa[i] = xa[i] * (0.54+0.46 * cos(i * pi/(lag-1))); }
      if ((mode != 1) && (mode != 3))
        { j = nfft - i;
          xa[j] = xa[i];
        }
```

```
    }
  nlast = nfft - lag;
  if ((mode == 1) || (mode == 3)) nlast = nfft;
  for (i=lag;i<=nlast;i++)
    { xa[i] = 0.0; }
  for (i=0;i<nfft;i++)
    { xc[i] = xa[i];
      xs[i] = 0.0;
    }
  fft(xc,xs,nfft,1);
  nhf1 = nfft/2 + 1;
  for (i=0;i<nhf1;i++)
    { freq[i] = i * fs/(double)nfft;
    if (sdb == 0)
      { sxy[i] = sqrt(xc[i] * xc[i]+xs[i] * xs[i]); }
      else
      { sxy[i] = 20.0 * log10(sqrt(xc[i] * xc[i]+xs[i] * xs[i])); }
    }
  free(xa);
  free(xc);
  free(xs);
  free(zc);
  free(zs);
}
```

五、例　题

下面给出主函数程序，它调用子函数 cmpse()。通过人机对话输入参数后，它能计算出相关函数和功率谱，同时将功率谱数据存入文件 powspec.dat 中。

例：信号 $x(i)$和 $y(i)$分别是频率为 f 赫兹的余弦和正弦函数

$$x(i) = \cos(2\pi i f/f_s)$$

$$y(i) = \sin(2\pi i f/f_s)$$

选择参数为：信号频率 f=1000Hz，采样频率 f_s=10000Hz，数据长度 n=256，每段长度 m=128，mode=2，海明窗函数，用于谱估计的相关函数点数 lag=32，用于谱估计的 FFT 长度 nfft=512，输出功率谱用 dB 表示。

主函数程序(文件名:cmpse.m)：

```
#include "stdio.h"
#include "math.h"
#include "cmpse.c"
```

```
main()
{ int i,m,n,sdb,lag,win,mode,nfft,len,nfft21;
  double fs,pi,r[128];
  static double x[512],y[512],freq[128],sxy[128];
  FILE *fp;
  len = 512;
  pi = 4.0 * atan(1.0);
  for (i=0;i<len;i++)
    { x[i] = cos(2 * pi * 0.1 * i); }
  for (i=0;i<len;i++)
    { y[i]=sin(2 * pi * 0.1 * i); }
  printf("\ninput the sample number n\n");
  scanf("%d",&n);
  printf("sampling frequency fs\n");
  scanf("%lf",&fs);
  printf("input section size m\n");
  scanf("%d",&m);
  printf("input the mode of correlation function\n");
  scanf("%d",&mode);
  printf("input the window type: 1 for rectangular; 2 for hamming\n");
  scanf("%d",&win);
  printf("input the number of correlation values used\n");
  scanf("%d",&lag);
  printf("input the size of FFT for spectral estimate\n");
  scanf("%d",&nfft);
  printf("input type of spectrum: 0 for linear, 1 for log spec in dB\n");
  scanf("%d",&sdb);
  cmpse(x,y,n,m,mode,win,fs,lag,nfft,r,freq,sxy,sdb);
  nfft21 = nfft/2 + 1;
  printf("\nCorrelation Function\n");
  for (i=0; i<nfft21; i++)
    { printf("%7d%11.3lf",i,r[i]);
      if ((i+1)%4 == 0) printf("\n");
    }
  printf("\nLog Power Spectrum:\n");
  printf("\n       FREQ          dB        FREQ          dB");
  printf("       FREQ          dB\n");
  for (i=0; i<nfft21; i++)
```

```
    { printf("%11.1lf%13.4lf",freq[i],sxy[i]);
      if ((i+1)%3 == 0) printf("\n");
    }
  fp = fopen("cmpse.dat","w");
  for (i=0; i<nfft21; i+-)
    { fprintf(fp,"%11.1lf%13.4lf\n",freq[i],sxy[i]);}
  fclose(fp);
}
```

运行结果:

相关函数(0~64)为

0	0.502	1	0.403	2	0.151	3	−0.156
4	−0.401	5	−0.492	6	−0.395	7	−0.148
8	0.153	9	0.393	10	0.482	11	0.387
12	0.145	13	−0.150	14	−0.386	15	−0.473
16	−0.379	17	−0.142	18	0.147	19	0.378
20	0.463	21	0.371	22	0.139	23	−0.144
24	−0.370	25	−0.453	26	−0.363	27	−0.136
28	0.141	29	0.362	30	0.443	31	0.356
32	0.133	33	−0.138	34	−0.354	35	−0.434
36	−0.348	37	−0.130	38	0.135	39	0.346
40	0.424	41	0.340	42	0.127	43	−0.132
44	−0.338	45	−0.414	46	−0.332	47	−0.124
48	0.129	49	0.330	50	0.404	51	0.324
52	0.121	53	−0.126	54	−0.322	55	−0.395
56	−0.316	57	−0.118	58	0.123	59	0.314
60	0.385	61	0.308	62	0.115	63	−0.120
64	−0.307						

信号 $x(i)$ 与 $y(i)$ 的互功率谱如下

FREQ	dB	FREQ	dB	FREQ	dB
0.0	−21.4638	78.1	−52.2276	156.2	−21.9576
234.4	−51.8240	312.5	−21.1400	390.6	−36.8523
468.8	−24.2255	546.9	−38.4523	625.0	−29.4264
703.1	−11.0800	781.2	3.9527	859.4	12.6999
937.5	17.1394	1015.6	18.1236	1093.8	15.8466
1171.9	9.9806	1250.0	−0.5382	1328.1	−17.6588
1406.2	−32.1344	1484.4	−42.3577	1562.5	−38.4651
1640.6	−25.3542	1718.8	−24.1901	1796.9	−36.6675
1875.0	−29.6832	1953.1	−29.4408	2031.2	−26.0116

2109.4	−38.0949	2187.5	−30.1633	2265.6	−32.3913
2343.8	−27.7095	2421.9	−39.9792	2500.0	−31.0731
2578.1	−34.7849	2656.2	−29.0480	2734.4	−41.9874
2812.5	−31.8907	2890.6	−36.8848	2968.8	−30.0795
3046.9	−44.1573	3125.0	−32.5536	3203.1	−38.8432
3281.2	−30.8686	3359.4	−46.6321	3437.5	−33.0713
3515.6	−40.7697	3593.8	−31.4657	3671.9	−49.6940
3750.0	−33.4645	3828.1	−42.7646	3906.2	−31.9073
3984.4	−54.0214	4062.5	−33.7518	4140.6	−44.9470
4218.8	−32.2190	4296.9	−62.3456	4375.0	−33.9474
4453.1	−47.4970	4531.2	−32.4180	4609.4	−67.1435
4687.5	−34.0610	4765.6	−50.7618	4843.8	−32.5150
4921.9	−55.6563	5000.0	−34.0983		

信号 x (i) 与 y (i) 的互功率谱如图 3—1—2 所示。

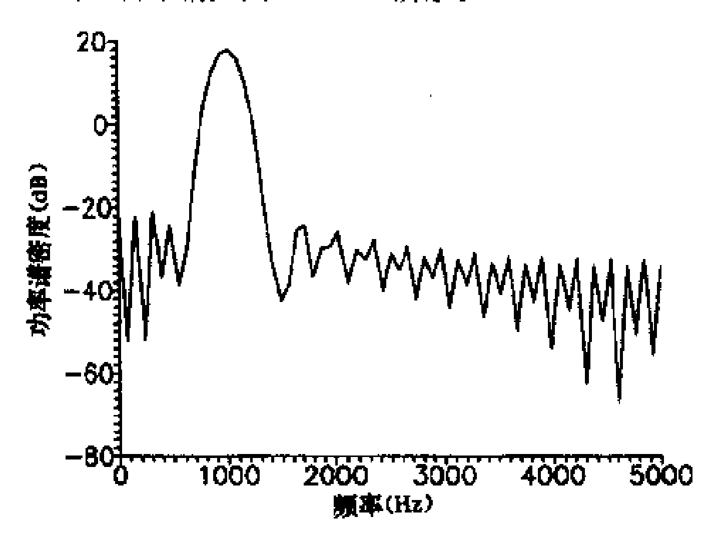


图 3—1—2 信号 $x(i)$与 $y(i)$的互功率谱

第二章　现代谱估计

§2.1　求解一般托布利兹方程组的莱文森算法

一、功　能

用莱文森(Levinson)算法求解对称托布利兹(Toeplitz)方程组。

二、方法简介

n 阶对称托布利兹方程组可以写成如下矩阵形式

$$\begin{bmatrix} t_0 & t_1 & t_2 & \cdots & t_{n-1} \\ t_1 & t_0 & t_1 & \cdots & t_{n-2} \\ t_2 & t_1 & t_0 & \cdots & t_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{n-1} & t_{n-2} & t_{n-3} & \cdots & t_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

求解对称托布利兹方程组的莱文森算法如下：

1. 初始化

$$a_0=t_0,\ y_0^{(1)}=1,\ x_0^{(1)}=b_0/t_0$$

2. 对于 $k=1, 2, \cdots, n-1$，作如下计算：

$$q_k=\sum_{j=0}^{k-1} x_j^{(k)} t_{k-j}$$

$$\beta_{k-1}=\sum_{j=0}^{k-1} y_j^{(k)} t_{j+1}$$

$$c_{k-1}=-\beta_{k-1}/a_{k-1}$$

$$\begin{cases} y_0^{(k+1)}=c_{k-1} y_{k-1}^{(k)} \\ y_i^{(k+1)}=y_{i-1}^{(k)}+c_{k-1} y_{k-i-1}^{(k)}, \quad i=1,2,\cdots,k-1 \\ y_k^{(k+1)}=y_{k-1}^{(k)} \end{cases}$$

$$a_k=a_{k-1}+c_{k-1}\beta_{k-1}$$

$$\omega_k=(b_k-q_k)/a_k$$

$$\begin{cases} x_i^{(k+1)} = x_i^{(k)} + \omega_k y_i^{(k+1)} \\ x_k^{(k+1)} = \omega_k y_k^{(k+1)} \end{cases}, \quad i=0,1,\cdots,k-1$$

三、使用说明

1. 子函数语句

int levin(t,b,n,x)

2. 形参说明

t ——双精度实型一维数组，长度为n。存放对称托布利兹矩阵的元素 $t_0, t_1, \cdots, t_{n-1}$。

b ——双精度实型一维数组，长度为n。存放方程组右端的常数向量。

n ——整型变量。方程组的阶数。

x ——双精度实型一维数组，长度为n。存放方程组的解。

本函数的返回值若小于0，则说明方程是病态的。

四、子函数程序(文件名:levin.c)

```
#include "stdlib.h"
int levin(t,b,n,x)
int n;
double t[],b[],x[];
{ int i,j,k;
  double a,beta,q,c,h, *y, *s;
  s = malloc(n * sizeof(double));
  y = malloc(n * sizeof(double));
  a = t[0];
  if ((fabs(a)+1.0) == 1.0)
    { free(s);
      free(y);
      printf("ill-conditioned\n");
      return(-1);
    }
  y[0] = 1.0;
  x[0] = b[0]/a;
  for (k=1;k<n;k++)
    { beta = 0.0;
      q = 0.0;
      for (j=0;j<k;j++)
        { beta += y[j] * t[j+1];
          q += x[j] * t[k-j];
```

```
        }
      if ((fabs(a)+1.0) == 1.0)
        { free(s);
          free(y);
          printf("ill-conditioned\n");
          return(-1);
        }
      c = -beta/a;
      s[0] = c * y[k-1];
      y[k] = y[k-1];
      if (k != 1)
        { for (i=1;i<k;i++)
            { s[i] = y[i-1] + c * y[k-i-1]; }
        }
      s[k] = y[k-1];
      a += c * beta;
      if ((fabs(a)+1.0) == 1.0)
        { free(s);
          free(y);
          printf("ill-conditioned\n");
          return(-1);
        }
      h = (b[k]-q)/a;
      for (i=0;i<k;i++)
        { x[i] += h * s[i];
          y[i] = s[i];
        }
      x[k] = h * y[k];
    }
  free(s);
  free(y);
  return(1);
}
```

五、例　题

求解六阶对称托布利兹方程组 $AX=B$，其中

$$A = \begin{bmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 6 & 5 & 4 & 3 & 2 \\ 4 & 5 & 6 & 5 & 4 & 3 \\ 3 & 4 & 5 & 6 & 5 & 4 \\ 2 & 3 & 4 & 5 & 6 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

即 $t=(6,5,4,3,2,1)$，常数向量 $B=(11,\ 9,\ 9,\ 9,\ 13,\ 17)^T$。

主函数程序(文件名:levin.m)：

```
#include "stdio.h"
#include "levin.c"
main()
{ int i;
  static double r[6]={6.0,5.0,4.0,3.0,2.0,1.0};
  static double b[6]={11.0,9.0,9.0,9.0,13.0,17.0};
  static double x[6];
  i = levin(r,b,6,x);
  if (i > 0)
    { for (i=0;i<6;i++)
        { printf("x(%d) = %10.7lf\n",i,x[i]); }
    }
}
```

运行结果：

方程组的解为

```
x(0) =  3.0000000
x(1) = -1.0000000
x(2) =  0.0000000
x(3) = -2.0000000
x(4) =  0.0000000
x(5) =  4.0000000
```

§2.2 求解对称正定方程组的乔里斯基算法

一、功　能

用乔里斯基(Cholesky)算法求解对称正定方程组。

二、方法简介

n 阶对称正定方程组的矩阵形式为

$$AX = B$$

其中 $A = (a_{ij})_{n\times n}$，$X = (x_i)_{n\times 1}$，$B = (b_i)_{n\times 1}$。

矩阵 A 的乔里斯基分解为

$$A = LDL^T$$

这里 D 是主对角元素都为正实数的对角阵，即 $D = \mathrm{diag}(d_1,d_2,\cdots,d_n)$，$d_i > 0$。$L$ 是主对角元素都为 1 的下三角阵

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}$$

用乔里斯基算法求解对称正定方程组的方法是，先用回代方法求解方程组 $LY = B$；得到 Y 后，再用回代方法求解方程 $L^TX = D^{-1}Y$。具体算法如下：

1. 将矩阵 A 进行乔里斯基分解

$$d_1 = a_{11}$$

对于 $i = 2,3,\cdots,n$，进行如下计算

$$l_{ij} = \begin{cases} \dfrac{a_{i1}}{d_1} , & j = 1 \\ \dfrac{a_{ij}}{d_j} - \displaystyle\sum_{k=1}^{j-1} \dfrac{l_{ik}d_kl_{jk}}{d_j} , & j = 2,3,\cdots,i-1 \end{cases}$$

$$d_i = a_{ii} - \sum_{k=1}^{i-1} d_k l_{ik}^2$$

2. 进行第一次回代，求解 Y

$$y_1 = b_1$$

$$y_k = b_k - \sum_{j=1}^{k-1} l_{kj}y_j \quad , \quad k = 2,3,\cdots,n$$

3. 进行第二次回代，求解 X

$$x_n = \frac{y_n}{d_n}$$

$$x_k = \frac{y_k}{d_k} - \sum_{j=k+1}^{n} l_{jk}x_j \quad , \quad k = n-1,n-2,\cdots,1$$

三、使用说明

1. 子函数语句

cholesky(a,b,n)

2. 形参说明

a ——双精度实型一维数组，长度为$\dfrac{n(n+1)}{2}$。存放对称正定矩阵 A 的下三角矩阵的元素，以行的顺序存放，即 $a_{00}, a_{10}, a_{11}, a_{20}, a_{21}, a_{22}, a_{30}, \cdots$。

b ——双精度实型一维数组，长度为 n。开始时存放方程组右端的常数向量，最后存

放方程组的解。

n ——整型变量。方程组的阶数。

对于病态方程组，本程序给出提示信息。

四、子函数程序(文件名:cholesky.c)

```
#include "stdlib.h"
#include "math.h"
void cholesky(a,b,n)
int n;
double a[],b[];
{ int i,j,k,m;
  double *d,*y,*xl,eps;
  d = malloc(n*sizeof(double));
  y = malloc(n*sizeof(double));
  xl = malloc(n*n*sizeof(double));
  eps = 1.0e-15;
  m = 0;
  d[0] = a[m];
  for (i=1;i<n;i++)
    { for (j=0;j<i;j++)
        { m = m + 1;
          xl[i*n+j] = a[m]/d[j];
          if (j == 0) continue;
          for (k=0;k<j;k++)
            { xl[i*n+j]-xl[i*n+k]*xl[j*n+k]*d[k]/d[j]; }
        }
      m = m + 1;
      d[i] = a[m];
      for (k=0;k<i;k++)
        { d[i] = d[i] - d[k]*xl[i*n+k]*xl[i*n+k]; }
      if ( fabs(d[i] ) < eps)
        { printf("\nill-conditioned ! \n");
          return;
        }
    }
  y[0] = b[0];
  for (k=1;k<n;k++)
    { y[k] = b[k];
```

```
        for (j=0;j<k;j++)
          { y[k] = y[k] - xl[k * n+j] * y[j]; }
    }
  b[n-1] = y[n-1]/d[n-1];
  for (k=(n-2);k>=0;k--)
    { b[k] = y[k]/d[k];
      for (j=(k+1);j<n;j++)
        { b[k] = b[k] - xl[j * n+k] * b[j]; }
    }
  free(d);
  free(y);
  free(xl);
}
```

五、例　题

求解四阶对称方程组 $AX=B$，其中

$$A=\begin{bmatrix}1 & 3 & 6 & 2\\ 3 & 2 & 5 & 3\\ 6 & 5 & 10 & 7\\ 2 & 3 & 7 & 5\end{bmatrix}$$

右端常数向量 $B=(19,\ 21,\ 47,\ 30)^T$。

主函数程序(文件名:cholesky.m):

```
#include "stdio.h"
#include "cholesky.c"
main()
{ int i,n;
  static double a[10] = {1, 3, 2, 6, 5, 10, 2, 3, 7, 5};
  static double b[4] = {19, 21, 47, 30};
  void cholesky();
  n = 4;
  cholesky(a,b,n);
  for (i=0;i<n;i++)
    { printf("        x(%d) = %10.7lf\n",i,b[i]); }
}
```

运行结果:

方程组的解为

```
        x(0) =   1.0000000
        x(1) =   2.0000000
```

```
x(2) =  1.0000000
x(3) =  3.0000000
```

§2.3 求解尤利-沃克方程的莱文森-德宾算法

一、功能

用莱文森-德宾(Levinson-Durbin)算法求解尤利-沃克(Yule-Walker)方程。

二、方法简介

p 阶尤利-沃克方程的矩阵形式为

$$\begin{bmatrix} r(0) & r(1) & \cdots & r(p) \\ r(1) & r(0) & \cdots & r(p-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(p) & r(p-1) & \cdots & r(0) \end{bmatrix} \begin{bmatrix} 1 \\ a(1) \\ \vdots \\ a(p) \end{bmatrix} = \begin{bmatrix} \rho \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

其中 $r(m)$是相关函数,$\{a(1),a(2),\cdots,a(p)\}$是 AR($p$)模型的系数,$\rho$ 是预测误差功率。

该方程实际上是本章§2.1节托布利兹方程的一种特殊情况,它可以用莱文森-德宾算法更加有效地求解。经过阶次递推,可依次计算出$\{a_1(1),\rho_1\}$,$\{a_2(1),a_2(2),\rho_2\}$,$\cdots$,$\{a_p(1),a_p(2),\cdots,a_p(p)\rho_p\}$,其中 AR 模型参量都加上了下标以表明阶数,阶数为 p 的最后那组参量就是方程的解。具体算法如下:

1. 初值化

$$a_1(1) = -\frac{r(1)}{r(0)}$$

$$\rho_1 = (1 - |a_1(1)|^2)\, r(0)$$

2. 对于 $k=2,3,\cdots,p$,进行如下计算

$$a_k(k) = -\frac{r(k) + \sum_{j=1}^{k-1} a_{k-1}(j) r(k-j)}{\rho_{k-1}}$$

$$a_k(i) = a_{k-1}(i) + a_k(k) a_{k-1}(k-i) \quad , \quad i=1,2,\cdots,k-1$$

$$\rho_k = (1 - |a_k(k)|^2)\rho_{k-1}$$

三、使用说明

1. 子函数语句

```
void levinson(r,p,a,v)
```

2. 形参说明

r ——双精度实型一维数组,长度为(p+1)。存放 Yule-Walker 方程的元素 r(0),r(1),$\cdots$,r(p)。

p ——整型变量。AR 模型的阶数。

a ——双精度实型一维数组,长度为 (p+1)。存放 AR 模型的系数 $a(0),a(1),\cdots,a(p)$。

v ——双精度实型指针。它指向预测误差功率 ρ，即 AR 模型激励白噪声的方差 σ^2。

四、子函数程序(文件名:levinson. c)

```
#include "stdlib.h"
void levinson(r,p,a,v)
int p;
double *v, a[],r[];
{ int i,k;
  double q, *b;
  b = malloc((p+1) * sizeof(double));
  a[0] = 1.0;
  a[1] = -r[1]/r[0];
  v[0] = (1 - a[1] * a[1]) * r[0];
  for (k=2;k<=p;k++)
    { q = 0.0;
      for (i=0;i<k;i++)
        { q += a[i] * r[k-i]; }
      a[k] = -q/v[0];
      for (i=1;i<k;i++)
        { b[i] = a[i] + a[k] * a[k-i]; }
      for (i=1;i<k;i++)
        { a[i] = b[i]; }
      v[0] = (1 - a[k] * a[k]) * v[0];
    }
  free(b);
}
```

五、例　题

已知实序列 $x(i)$ 的自相关为 $r_{xx}(0)=55$，$r_{xx}(1)=40$，$r_{xx}(2)=26$，$r_{xx}(3)=14$，用莱文森-德宾算法求其 AR 模型参数 $\{a(1),a(2),a(3),\rho\}$。

主函数程序(文件名:levinson. m):

```
#include "stdio.h"
#include "levinson.c"
main()
{ int i,p;
  double v,a[4];
  static double r[4] = {55,40,26,14};
  p = 3;
```

```
    levinson(r,p,a,&v);
    printf("The Coefficients of AR Model\n");
    for (i=0;i<4;i++)
      { printf("a(%d) = %10.7lf\n",i,a[i]); }
    printf("The Prediction Error Power for the Predictor \n");
    printf("Pe = %10.7lf\n",v);
  }
```

运行结果：

AR 模型的系数为

```
            a(0) =   1.0000000
            a(1) = -0.8028575
            a(2) =   0.0430279
            a(3) =   0.0936942
```

预测误差功率为

```
            Pe =   25.3161423
```

§2.4 计算 ARMA 模型的功率谱密度

一、功　能

计算 ARMA(p,q)模型的功率谱密度。

二、方法简介

平稳随机序列 $x(n)$的 ARMA(p,q)模型为

$$x(n)+\sum_{i=1}^{p}a(i)x(n-i)=\sum_{i=0}^{q}b(i)w(n-i)$$

其中 $a(i)(i=1,2,\cdots,p)$和 $b(i)(i=0,1,\cdots q)$分别是 ARMA 模型的自回归(AR)参数和滑动平均(MA)参数，$w(n)$是均值为零、方差为 σ^2 的白噪声序列。

ARMA(p,q)模型的功率谱密度为

$$S(\omega)=\sigma^2\left|\frac{\sum_{i=0}^{q}b(i)\mathrm{e}^{-j\omega i}}{1+\sum_{i=1}^{p}a(i)\mathrm{e}^{-j\omega i}}\right|^2$$

三、使用说明

1. 子函数语句

void psd(b,a,q,p,sigma2,fs,x,freq,len,sign)

2. 形参说明

b ——双精度实型一维数组，长度为$(q+1)$。存放 ARMA(p,q)模型的滑动平均系

数。

a ——双精度实型一维数组，长度为($p+1$)。存放 ARMA(p,q)模型的自回归系数。

q ——整型变量。ARMA(p,q)模型的滑动平均阶数。

p ——整型变量。ARMA(p,q)模型的自回归阶数。

sigma2 ——双精度实型变量。ARMA(p,q)模型白噪声激励的方差 σ^2。

fs ——双精度实型变量。采样频率(Hz)。

x ——双精度实型一维数组。长度为 len。当 sign=0 时，存放功率谱密度；当 sign=1 时，存放用分贝表示的功率谱密度。

freq ——双精度实型一维数组，长度为 len。存放功率谱密度所对应的频率。

len——整型变量，功率谱密度的数据点数。

sign ——整型变量。当 sign=0 时，计算功率谱密度；当 sign=1 时，计算用分贝表示的功率谱密度。

四、子函数程序(文件名:psd.c)

```
#include "math.h"
void psd(b,a,q,p,sigma2,fs,x,freq,len,sign)
int p,q,len,sign;
double fs,sigma2,b[], a[], x[], freq[];
{ int i,k;
  double ar,ai,br,bi,zr,zi,im,re,xre,xim;
  double ang,den,numr,numi,temp;
  for (k=0;k<len;k++)
    { ang = k * 0.5/(len-1);
      freq[k] = ang * fs;
      zr = cos(-8.0 * atan(1.0) * ang);
      zi = sin(-8.0 * atan(1.0) * ang);
      br = 0.0;
      bi = 0.0;
      for (i=q;i>0;i--)
        { re = br;
          im = bi;
          br = ( re + b[i] ) * zr - im * zi;
          bi = ( re + b[i] ) * zi + im * zr;
        }
      ar = 0.0;
      ai = 0.0;
      for (i=p;i>0;i --)
        { re = ar;
```

```
        im = ai;
        ar = ( re + a[i] ) * zr - im * zi;
        ai = ( re + a[i] ) * zi + im * zr;
      }
    br = br + b[0];
    ar = ar + 1.0;
    numr = ar * br + ai * bi;
    numi = ar * bi - ai * br;
    den = ar * ar + ai * ai;
    xre = numr/den;
    xim = numi/den;
    switch (sign)
      { case 0:
        { x[k] = xre * xre + xim * xim;
          x[k] = sigma2 * x[k]/fs;
          break;
        }
      case 1:
        { temp = xre * xre + xim * xim;
          temp = sigma2 * temp/fs;
          if (temp == 0.0) temp = 1.0e-20;
          x[k] = 10.0 * log10(temp);
        }
      }
  }
}
```

五、例　题

AR(4)模型为

$$x(n)-2.7607x(n-1)+3.8106x(n-2)-2.6535x(n-3)+0.9238x(n-4)=w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。计算 AR 序列的功率谱密度。

主函数程序(文件名:psd.m):

```
#include "stdio.h"
main()
{ int i,len;
  static double a[5]={1, -2.7607, 3.8106, -2.6535, 0.9238}, b[1]={1};
```

```
    static double fs,sigma2,x[300],freq[300];
    FILE *fp;
    void psd();
    fs = 1.0;
    sigma2 = 1.0;
    len = 300;
    psd(b,a,0,4,sigma2,fs,x,freq,len,0);
    if ( ( fp = fopen("psd.dat","w")) == NULL )
      { printf("cannot open file 'psd.dat' ! \n");
        exit(0);
      }
    for(i=0;i<300;i++)
      { fprintf(fp," %lf        %lf\n",freq[i],x[i]); }
    fclose(fp);
  }
```

运行结果：

AR(4)模型的功率谱密度如图 3－2－1 所示。

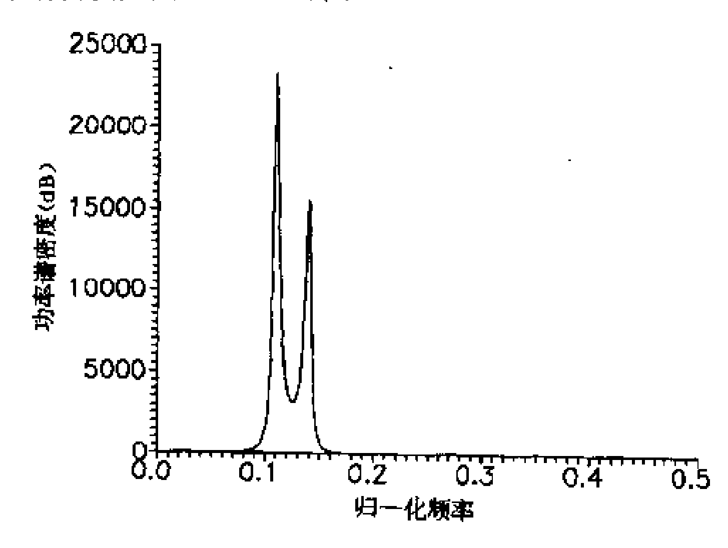


图 3－2－1　AR(4)模型的功率谱

§2.5　尤利-沃克谱估计算法

一、功　能

用自相关方法估计 AR 模型参数，进而实现功率谱估计。该方法也称为尤利-沃克(Yule-Walker)算法。

二、方法简介

平稳随机序列 $x(n)(n=0,1,\cdots,N-1)$的 AR(p)模型为

$$x(n)+\sum_{i=1}^{p}a(i)\,x(n-i)=w(n)$$

其中 $a(i)(i=1,2,\cdots,p)$是 AR 系数，$w(n)$是均值为零、方差为 σ^2 的白噪声。

用自相关法估计 AR 模型参数的具体步骤如下：

1. 计算自相关函数

$$r_{xx}(k)=\frac{1}{N}\sum_{n=0}^{N-1-k}x(n)\,x(n+k)\quad,\quad k=0,1,\cdots,p$$

2. 用莱文森-德宾算法求解尤利-沃克方程

$$\begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(p) \\ r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(p-1) \\ \vdots & \vdots & \ddots & \vdots \\ r_{xx}(p) & r_{xx}(p-1) & \cdots & r_{xx}(0) \end{bmatrix}\begin{bmatrix} 1 \\ a(1) \\ \vdots \\ a(p) \end{bmatrix}=\begin{bmatrix} \sigma^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

用 AR 参数的估计值，可以计算功率谱密度

$$S(\omega)=\frac{\sigma^2}{\left|1+\sum_{i=1}^{p}a(i)e^{-j\omega i}\right|^2}$$

三、使用说明

1. 子函数语句

void yulewalk(x,n,p,a,v)

2. 形参说明

x ——双精度实型一维数组，长度为 n。存放随机序列。

n ——整型变量。随机序列的长度。

p ——整型变量。AR 模型的阶数。

a ——双精度实型一维数组，长度为 (p+1)。存放 AR 模型的系数 $a(0),a(1),\cdots,a(p)$。

v ——双精度实型指针。它指向预测误差功率 σ^2，即 AR 模型激励白噪声的方差。

四、子函数程序(文件名:yulewalk.c)

```
#include "stdlib.h"
#include "levinson.c"
void yulewalk(x,n,p,a,v)
int n,p;
double *v,a[],x[];
{ int i,k;
  double s,*r;
```

```
  r = malloc((p+1) * sizeof(double));
  for (k=0;k<=p;k++)
    { s = 0.0;
      for (i=0;i<(n-k);i++)
        { s += x[i] * x[i+k]; }
      r[k] = s/n;
    }
  levinson(r,p,a,v);
  free(r);
}
```

五、例　题

例 1：用程序产生 40 点数据，它是下面 3 阶 IIR 数字滤波器的单位冲激响应

$$1 / [1 - 1.44z^{-1} + 1.26z^{-2} - 0.81z^{-3}]$$

用尤利-沃克方法估计该滤波器的系数。

主函数程序(文件名：yulewal1.m)：

```
#include "stdio.h"
#include "yulewalk.c"
main()
{ int i,n,p;
  double v,x[40];
  double a[4];
  n = 40;
  p = 3;
  x[0] = 1.0;
  x[1] = 1.44;
  x[2] = 1.44 * x[1] - 1.26;
  for (i=3;i<n;i++)
    { x[i] = 1.44 * x[i-1] - 1.26 * x[i-2]+0.81 * x[i-3];}
  yulewalk(x,n,p,a,&v);
  printf("The Coefficients of AR Model\n");
  for (i=0;i<=p;i++)
    { printf("a(%d) = %10.7lf\n",i,a[i]); }
  printf("The Prediction Error Power of AR Model\n");
  printf("Pe = %10.7lf\n",v);
}
```

运行结果：

AR 模型的系数为

```
a(0) =    1.0000000
a(1) = -1.3630849
a(2) =    1.1082684
a(3) = -0.7100407
```

预测误差功率为

```
Pe =    0.0382279
```

例 2：AR(4)模型如下

$$x(n) - 1.352x(n-1) + 1.338x(n-2) - 0.662x(n-3) + 0.24x(n-4) = w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个宽带随机过程。用尤利-沃克方法估计 AR 参数，并进行谱估计。

主函数程序(文件名：yulewal2.m)：

```
#include "stdio.h"
#include "arma.c"
#include "yulewalk.c"
#include "psd.c"
main()
{ int i,n,p,q,len;
  long seed;
  double v,mean,var,c[10],x[500],freq[200];
  double fs,sigma2;
  static double a[5] = {1.0, -1.352, 1.338, -0.662, 0.24};
  static double b[1] = {1.0};
  FILE *fp;
  p = 4;
  q = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 500;
  arma(a,b,p,q,mean,var,&seed,x,n);
  for (i=0;i<300;i++)
    x[i] = x[i+200];
  n = 300;
  yulewalk(x,n,p,c,&v);
  printf("The Coefficients of AR Model\n");
  for (i=0;i<=p;i++)
    { printf("a(%d) = %10.7lf\n",i,c[i]); }
  printf("The Prediction Error Power of AR Model\n");
```

```
    printf("Pe = %10.7lf\n",v);
    fs = 1.0;
    sigma2 = v;
    len = 200;
    psd(b,c,q,p,sigma2,fs,x,freq,len,1);
    fp = fopen("yulewal2.dat","w");
    for (i=0;i<len;i++)
      fprintf(fp,"%lf       %lf\n",freq[i],x[i]);
    fclose(fp);
  }
```

运行结果：

AR 模型的系数为

a(0) = 1.0000000

a(1) = −1.3323805

a(2) = 1.3059350

a(3) = −0.5808768

a(4) = 0.2067898

预测误差功率为

Pe = 1.0995336

其功率谱如图 3−2−2 所示。

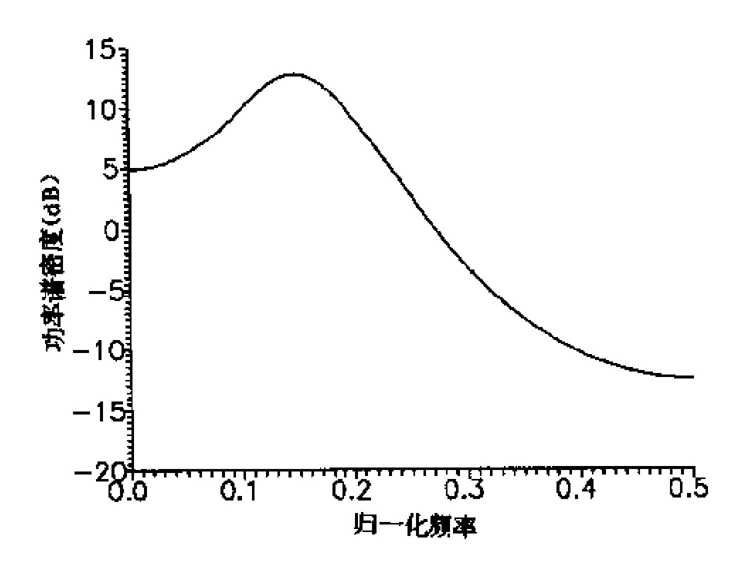


图 3−2−2　宽带 AR(3)模型的功率谱

§2.6 协方差谱估计算法

一、功 能

用协方差方法估计 AR 模型参数，进而实现功率谱估计。

二、方法简介

平稳随机序列 $x(n)(n=0,1,\cdots,N-1)$的 AR(p)模型为

$$x(n)+\sum_{i=1}^{p}a(i)x(n-i)=w(n)$$

其中 $a(i)(i=1,2,\cdots,p)$是 AR 系数，$w(n)$是均值为零、方差为 σ^2 的白噪声。

用协方差法估计 AR 模型参数的具体步骤如下：

1. 计算协方差

$$c_{xx}(j,k)=\frac{1}{N-p}\sum_{n=p}^{N-1}x(n-j)x(n-k)\quad,\quad j,k=0,1,\cdots,p$$

2. 用乔里斯基算法求解对称正定方程组

$$\begin{bmatrix}c_{xx}(1,1) & c_{xx}(1,2) & \cdots & c_{xx}(1,p)\\ c_{xx}(2,1) & c_{xx}(2,2) & \cdots & c_{xx}(2,p)\\ \vdots & \vdots & \ddots & \vdots\\ c_{xx}(p,1) & c_{xx}(p,2) & \cdots & c_{xx}(p,p)\end{bmatrix}\begin{bmatrix}a(1)\\ a(2)\\ \vdots\\ a(p)\end{bmatrix}=-\begin{bmatrix}c_{xx}(1,0)\\ c_{xx}(2,0)\\ \vdots\\ c_{xx}(p,0)\end{bmatrix}$$

3. 计算激励白噪声的方差

$$\sigma^2=c_{xx}(0,0)+\sum_{k=1}^{p}a(k)c_{xx}(0,k)$$

用 AR 模型参数的估计值，可以计算功率谱密度

$$S(\omega)=\frac{\sigma^2}{\left|1+\sum_{i=1}^{p}a(i)e^{-j\omega i}\right|^2}$$

三、使用说明

1. 子函数语句

void covar(x,n,p,a,v,mode)

2. 形参说明

x ——双精度实型一维数组，长度为 n。存放随机序列。

n ——整型变量。随机序列的长度。

p ——整型变量。AR 模型的阶数。

a ——双精度实型一维数组，长度为(p+1)。存放 AR 模型的系数 $a(0),a(1),\cdots,a(p)$。

v ——双精度实型指针。它指向预测误差功率 σ^2，即 AR 模型激励白噪声的方差。

mode ——整型变量。mode = 0，表示协方差算法；mode = 1，表示修正协方差算法。

四、子函数程序(文件名:covar.c)

```
#include "stdlib.h"
#include "cholesky.c"
void covar(x,n,p,a,v,mode)
int n,p,mode;
double *v,a[],x[];
{ int i,j,k,m;
  double cc,sum,*c;
  c = malloc((p*(p+1)/2)*sizeof(double));
  m = 0;
  for (k=1;k<=p;k++)
  for (j=1;j<=k;j++)
    { c[m] = 0.0;
      for (i=p;i<n;i++)
        { c[m] += x[i-j]*x[i-k]; }
      if (mode ==1)
        { for (i=0;i<(n-p);i++)
             { c[m] += x[i+j]*x[i+k]; }
        }
      m = m + 1;
    }
   for (j=1;j<=p;j++)
    { a[j-1] = 0.0;
      for (i=p;i<n;i++)
        { a[j-1] -= x[i-j]*x[i]; }
      if (mode ==1)
        { for (i=0;i<(n-p);i++)
             { a[j-1] -= x[i+j]*x[i]; }
        }
    }
   cholesky(c,a,p);
   for (k=(p-1);k>=0;k--)
     { a[k+1] = a[k]; }
   a[0] = 1.0;
   sum = 0.0;
```

```
  for (k=0;k<=p;k++)
    { cc = 0.0;
      for (i=p;i<n;i++)
        { cc += x[i] * x[i-k]; }
      if (mode ==1)
        { for (i=0;i<(n-p);i++)
            { cc += x[i] * x[i+k]; }
        }
      if (k == 0)
        { sum += cc; }
      else
        { sum += cc * a[k]; }
    }
  if ( mode == 1 )
    { v[0] = sum/(2 * (n-p)); }
  else
    { v[0] = sum/(n-p); }
  free(c);
}
```

五、例　题

例 1：AR(4)模型如下

$$x(n)-1.352x(n-1)+1.338x(n-2)-0.662x(n-3)+0.24x(n-4)=w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个宽带随机过程。用协方差法估计 AR 参数，并进行谱估计。

主函数程序(文件名：covar1.m)：

```
#include "stdio.h"
#include "arma.c"
#include "covar.c"
#include "psd.c"
main()
{ int i,n,p,q,len;
  long seed;
  double v,mean,var,c[10],x[500],freq[200];
  double fs,sigma2;
  static double a[5] = {1.0, -1.352, 1.338, -0.662, 0.24};
  static double b[1] = {1.0};
  FILE *fp;
```

```
    p = 4;
    q = 0;
    seed = 13579l;
    mean = 0.0;
    var = 1.0;
    n = 500;
    arma(a,b,p,q,mean,var,&seed,x,n);
    for (i=0;i<300;i++)
      x[i] = x[i+200];
    n = 300;
    covar(x,n,p,c,&v,0);
    printf("The coefficients of AR model\n");
    for (i=0;i<=p;i++)
      { printf("a(%d) = %10.7lf\n",i,c[i]); }
    printf("The reflect coefficients of AR model\n");
    printf("Pe = %10.7lf\n",v);
    fs = 1.0;
    sigma2 = v;
    len = 200;
    psd(b,c,q,p,sigma2,fs,x,freq,len,1);
    fp = fopen("covar1.dat","w");
    for (i=0;i<len;i++)
      fprintf(fp,"%lf      %lf\n",freq[i],x[i]);
    fclose(fp);
  }
```

运行结果：

AR 模型的系数为

a(0) = 1.0000000
a(1) = −1.3991396
a(2) = 1.4286759
a(3) = −0.6962182
a(4) = 0.2587227

预测误差功率为

Pe = 0.9993630

其功率谱如图 3−2−3 所示。

例 2：AR(4)模型如下

$$x(n) - 2.76x(n-1) + 3.809x(n-2) - 2.654x(n-3) + 0.924x(n-4) = w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个窄带随机过程。用协方差方法估计 AR

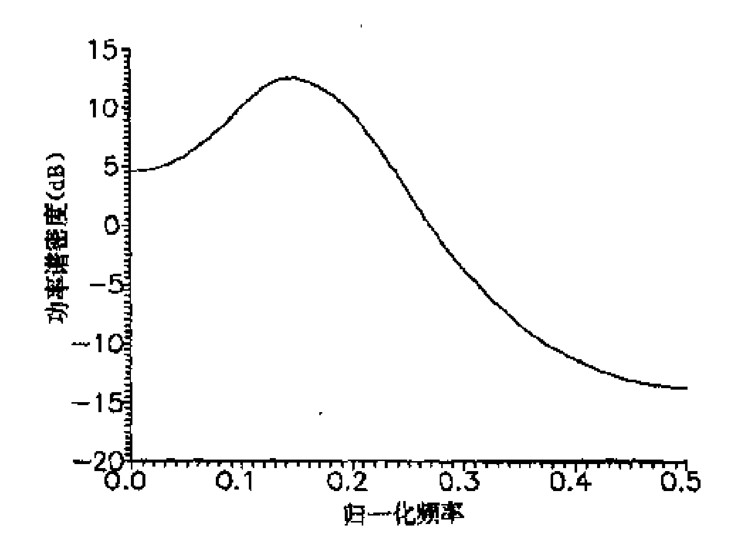


图 3—2—3　宽带 AR(4)模型的功率谱

参数，并进行谱估计。

主函数程序(文件名:covar2.m):

```
#include "stdio.h"
#include "arma.c"
#include "covar.c"
#include "psd.c"
main()
{ int i,n,p,q,len;
  long seed;
  double v,mean,var,c[10],x[300],freq[200];
  double fs,sigma2;
  static double a[5] = {1.0, -2.76, 3.809, -2.654, 0.924};
  static double b[1] = {1.0};
  FILE *fp;
  p = 4;
  q = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 500;
  arma(a,b,p,q,mean,var,&seed,x,n);
  for (i=0;i<300;i++)
    x[i] = x[i+200];
  n = 300;
```

```
    covar(x,n,p,c,&v,0);
    printf("The coefficients of AR model\n");
    for (i=0;i<=p;i++)
      { printf("a(%d) = %10.7lf\n",i,c[i]); }
    printf("The reflect coefficients of AR model\n");
    printf("Pe = %10.7lf\n",v);
    fs = 1.0;
    sigma2 = v;
    len = 200;
    psd(b,c,q,p,sigma2,fs,x,freq,len,1);
    fp = fopen("covar2.dat","w");
    for (i=0;i<len;i++)
      fprintf(fp," %lf      %lf\n",freq[i],x[i]);
    fclose(fp);
  }
```

运行结果：

AR 模型的系数为

a(0) = 1.0000000

a(1) = −2.7310949

a(2) = 3.7478402

a(3) = −2.5951549

a(4) = 0.9022404

预测误差功率为

Pe = 1.0055264

其功率谱如图 3−2−4 所示。

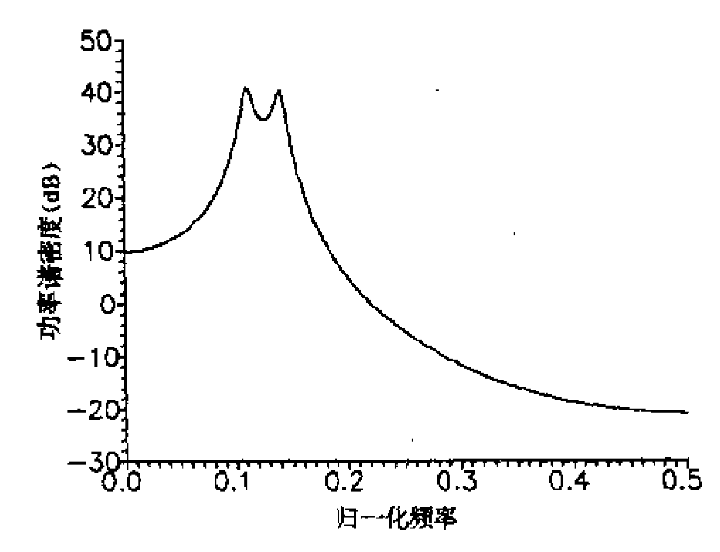


图 3−2−4　窄带 AR(4)模型的功率谱

§ 2.7　Burg 谱估计算法

一、功能

用 Burg 算法估计 AR 模型参数，进而实现功率谱估计。

二、方法简介

平稳随机序列 $x(n)(n=0,1,\cdots,N-1)$ 的 AR(p)模型为

$$x(n) + \sum_{i=1}^{p} a(i)\, x(n-i) = w(n)$$

其中 $a(i)(i=1,2,\cdots,p)$ 是 AR 系数，$w(n)$ 是均值为零、方差为 σ^2 的白噪声。

用 Burg 算法估计 AR 模型参数的具体步骤如下：

1. 初始化

$$r_{xx}(0) = \frac{1}{N}\sum_{n=0}^{N-1} |x(n)|^2$$

$$\rho_0 = \hat{r}_{xx}(0)$$

$$e_0^f(n) = x(n) \quad , \quad n=1,2,\cdots,N-1$$

$$e_0^b(n) = x(n) \quad , \quad n=0,1,\cdots,N-2$$

2. 对于 $k=1,2,\cdots,p$，进行如下计算，从而得到 AR 参数

$$a_k(k) = \frac{-2\sum_{n=k}^{N-1} e_{k-1}^f(n) e_{k-1}^b(n-1)}{\sum_{n=k}^{N-1}\left(|e_{k-1}^f(n)|^2 + |e_{k-1}^b(n-1)|^2\right)}$$

$$a_k(i) = a_{k-1}(i) + a_k(k)\, a_{k-1}(k-i) \quad , \quad i=1,2,\cdots,k-1$$

$$\rho_k = (1 - |a_k(k)|^2)\, \rho_{k-1}$$

$$e_k^f(n) = e_{k-1}^f(n) + a_k(k)\, e_{k-1}^b(n-1), n=k+1,k+2,\cdots,N-1$$

$$e_k^b(n) = e_{k-1}^b(n-1) + a_k(k) e_{k-1}^f(n), n=k,k+1,\cdots,N-2$$

其中预测误差功率 ρ 等于激励白噪声的方差 σ^2。

用 AR 模型参数的估计值，可以计算功率谱密度

$$S(\omega) = \frac{\sigma^2}{\left|1+\sum_{i=1}^{p} a(i) e^{-j\omega i}\right|^2}$$

三、使用说明

1. 子函数语句

```
void burg(x,n,p,a,v)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。存放随机序列。

n ——整型变量。随机序列的长度。

p ——整型变量。AR 模型的阶数。

a ——双精度实型一维数组，长度为(p+1)。存放 AR 模型的系数 $a(0), a(1), \cdots, a(p)$。

v —— 双精度实型指针。它指向预测误差功率 ρ，即 AR 模型激励白噪声的方差 σ^2。

四、子函数程序(文件名:burg.c)

```
#include "stdlib.h"
void burg(x,n,p,a,v)
int n,p;
double  *v, a[ ],x[ ];
{ int i,k;
  double r0,sumd,sumn, *b, *ef, *eb;
  b = malloc((p+1) * sizeof(double));
  ef = malloc(n * sizeof(double));
  eb = malloc(n * sizeof(double));
  a[0] = 1.0;
  r0 = 0.0;
  for (i=0;i<n;i++)
    { r0 += x[i] * x[i]/n; }
  v[0] = r0;
  for (i=1;i<n;i++)
    { ef[i] = x[i];
      eb[i-1] = x[i-1];
    }
  for (k=1;k<=p;k++)
    { sumn = 0.0;
      sumd = 0.0;
      for (i=k;i<n;i++)
        { sumn += ef[i] * eb[i-1];
          sumd += ef[i] * ef[i] + eb[i-1] * eb[i-1];
        }
      a[k] = -2 * sumn/sumd;
      for (i=1;i<k;i++)
        { b[i] = a[i] + a[k] * a[k-i]; }
      for (i=1;i<k;i++)
        { a[i] = b[i]; }
      v[0] = (1.0 - a[k] * a[k]) * v[0];
```

```
        for (i=(n-1);i>=(k+1);i--)
          { ef[i] = ef[i] + a[k] * eb[i-1];
            eb[i-1] = eb[i-2] + a[k] * ef[i-1];
          }
      }
    free(b);
    free(ef);
    free(eb);
  }
```

五、例 题

例 1：AR(4)模型如下

$$x(n) - 1.352x(n-1) + 1.338x(n-2) - 0.662x(n-3) + 0.24x(n-4) = w(n)$$

其中 $w(n)$ 是零均值、单位方差的白噪声。这是一个宽带随机过程。用 Burg 算法估计 AR 参数，并进行谱估计。

主函数程序(文件名：burg1.m)：

```
  #include "stdio.h"
  #include "arma.c"
  #include "burg.c"
  #include "psd.c"
  main()
  { int i,n,p,q,len;
    long seed;
    double v,fs,mean,var,sigma2;
    double c[20],x[500],freq[200];
    static double a[5] = {1.0, -1.352, 1.338, -0.662, 0.24};
    static double b[1] = {1.0};
    FILE *fp;
    p = 4;
    q = 0;
    seed = 13579l;
    mean = 0.0;
    var = 1.0;
    n = 500;
    arma(a,b,p,q,mean,var,&seed,x,n);
    for (i=0;i<300;i++)
      x[i] = x[i+200];
```

```
    n = 300;
    burg(x,n,p,c,&v);
    printf("The Coefficients of AR model\n");
    for (i=0;i<=p;i++)
      { printf("a(%d) = %10.7lf\n",i,c[i]); }
    printf("The Prediction Error Power of AR model\n");
    printf("pe = %10.7lf\n",v);
    len = 200;
    fs = 1.0;
    sigma2 = v;
    psd(b,c,q,p,sigma2,fs,x,freq,len,1);
    fp = fopen("burg1.dat","w");
    for (i=0;i<len;i++)
      fprintf(fp,"%lf %lf\n",freq[i],x[i]);
    fclose(fp);
  }
```

运行结果：

AR 模型的系数为

$$a(0) = 1.0000000$$
$$a(1) = -1.3990654$$
$$a(2) = 1.4266702$$
$$a(3) = -0.6989958$$
$$a(4) = 0.2611651$$

预测误差功率为

$$Pe = 1.0116348$$

其功率谱如图 3－2－5 所示。

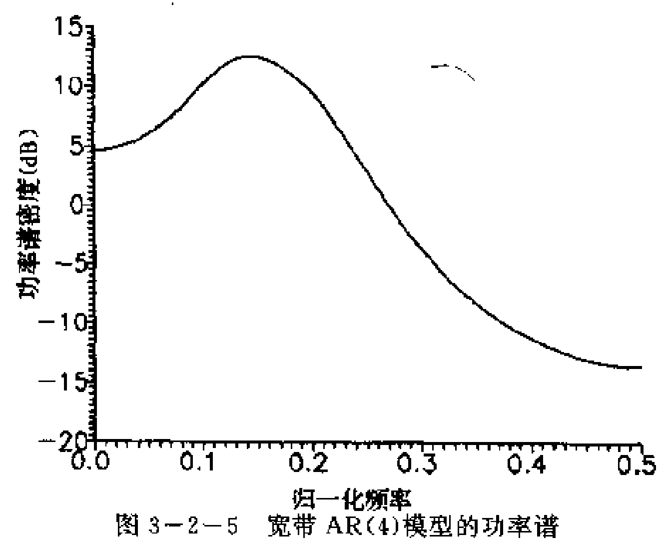


图 3－2－5　宽带 AR(4)模型的功率谱

例 2：AR(4)模型如下

$$x(n) - 2.76x(n-1) + 3.809x(n-2) - 2.654x(n-3) + 0.924x(n-4) = w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个窄带随机过程。用 Burg 算法估计 AR 参数，并进行谱估计。

主函数程序(文件名：burg2.m)：

```
#include "stdio.h"
#include "arma.c"
#include "burg.c"
#include "psd.c"
main()
{ int i,n,p,q,len;
  long seed;
  double v,fs,mean,var,sigma2;
  double c[20],x[500],freq[200];
  static double a[5] = {1.0, -2.76, 3.809, -2.654, 0.924};
  static double b[1] = {1.0};
  FILE *fp;
  p = 4;
  q = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 500;
  arma(a,b,p,q,mean,var,&seed,x,n);
  for (i=0;i<300;i++)
    x[i] = x[i+200];
  n = 300;
  burg(x,n,p,c,&v);
  printf("The Coefficients of AR model\n");
  for (i=0;i<=p;i++)
    { printf("a(%d) = %10.7lf\n",i,c[i]); }
  printf("The Prediction Error Power of AR model\n");
  printf("Pe = %10.7lf\n",v);
  len = 200;
  fs = 1.0;
  sigma2 = v;
  psd(b,c,q,p,sigma2,fs,x,freq,len,1);
  fp = fopen("burg2.dat","w");
```

```
    for (i=0;i<len;i++)
      fprintf(fp," %lf      %lf\n",freq[i],x[i]);
    fclose(fp);
  }
```

运行结果：

AR 模型的系数为

a(0) = 1.0000000

a(1) = −2.7374704

a(2) = 3.7566198

a(3) = −2.6013084

a(4) = 0.9025563

预测误差功率为

Pe = 1.0109662

其功率谱如图 3—2—6 所示。

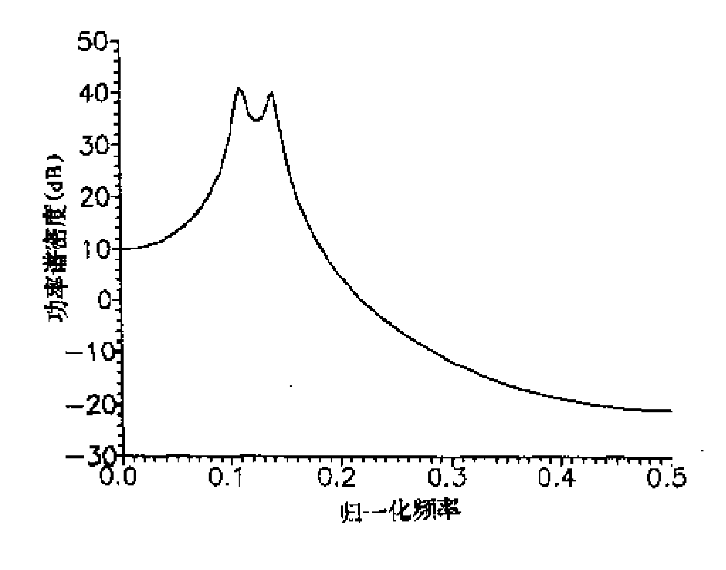


图 3—2—6 窄带 AR(4)模型的功率谱

§2.8 最大似然谱估计算法

一、功 能

用 Capon 提出的最大似然方法进行功率谱估计。该算法实际上是对功率谱作无偏最小方差估计，它比 Burg 谱估计算法分辨率稍低，但更加稳定。

二、方法简介

设随机序列为 $x(n)(0,1,\cdots,N-1)$，其自相关矩阵为 R_x，则在频率 ω 处的最大似然

功率谱估计为

$$S_{ML}(\omega) = \frac{1}{E^T(\omega)R_x^{-1}E(\omega)}$$

其中

$$E(\omega) = [1, e^{j\omega}, e^{j2\omega}, e^{j(N-1)\omega}]^T$$

当用最大似然法进行AR模型的功率谱估计时，可以证明：最大似然谱估计与AR模型谱估计的关系为

$$\frac{1}{S_{ML}(\omega)} = \sum_{i=0}^{p}\frac{1}{S_{AR}^{(i)}(\omega)}$$

该式表明，最大似然谱估计的倒数等于 $i=0,1,\cdots,p$ 阶 AR(i)模型谱估计的倒数和，从而可以用Burg谱估计算法来计算最大似然谱估计。最大似然谱估计比Burg谱估计分辨率稍低，但更加稳定。

三、使用说明

1. 子函数语句

void mlm(x,n,pm,fs,s,freq,len,sdb)

2. 形参说明

x ——双精度实型一维数组，长度为n。存放随机序列。

n ——整型变量。随机序列的长度。

pm ——整型变量。进行最大似然谱估计时所用AR模型的最大阶数。

fs ——双精度实型变量。信号的采样频率。

s ——双精度实型一维数组，长度为len。存放最大似然功率谱。

freq ——双精度实型一维数组，长度为len。存放与最大似然功率谱相对应的频率值。

len ——整型变量。功率谱密度的数据点数。

sdb ——整型变量。用来指明输出功率谱的形式。sdb = 0，线性功率谱；sdb = 1，用分贝表示的功率谱。

四、子函数程序(文件名:mlm.c)

```
#include "math.h"
#include "stdlib.h"
#include "burg.c"
void mlm(x,n,pm,fs,s,freq,len,sdb)
int n,pm,len,sdb;
double fs,x[],s[],freq[];
{ int i,k,p;
  double v,ai,ar,im,re,zi,zr,sar,ang,tpi, *a;
  a = malloc((pm+1) * sizeof(double));
```

```
tpi = 8.0 * atan(1.0);
for (i=0;i<len;i++)
  { s[i] = 0.0; }
for (p=0;p<=pm;p++)
  { burg(x,n,p,a,&v);
    for (k=0;k<len;k++)
      { ang = k * 0.5/(len-1);
        freq[k] = ang * fs;
        zr = cos(-tpi * ang);
        zi = sin(-tpi * ang);
        ar = 0.0;
        ai = 0.0;
        for (i=p;i>0;i--)
          { re = ar;
            im = ai;
            ar = ( re + a[i] ) * zr - im * zi;
            ai = ( re + a[i] ) * zi + im * zr;
          }
        ar = ar + 1.0;
        sar = v/(fs * (ar * ar + ai * ai));
        s[k] = s[k] + 1.0/sar;
      }
  }
for (k=0;k<len;k++)
  { s[k] = pm/s[k]; }
if (sdb == 1)
  { for (k=0;k<len;k++)
      { s[k] = 10.0 * log10(s[k]); }
  }
free(a);
}
```

五、例　题

例 1:AR(4)模型如下

$$x(n) - 1.352x(n-1) + 1.338x(n-2) - 0.662x(n-3) + 0.24x(n-4) = w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个宽带随机过程。用最大似然方法进行功率谱估计。

主函数程序(文件名:mlm1.m):

```
#include "stdio.h"
#include "arma.c"
#include "mlm.c"
main()
{ int i,n,p,q,pm,len,sdb;
  long seed;
  double fs,mean,var;
  static double x[500],freq[100],s[100];
  static double a[5] = {1.0, -1.352, 1.338, -0.662, 0.24};
  static double b[1] = {1.0};
  FILE *fp;
  p = 4;
  q = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 500;
  arma(a,b,p,q,mean,var,&seed,x,n);
  for (i=0;i<300;i++)
    { x[i] = x[i+200]; }
  n = 300;
  len = 100;
  fs = 1.0;
  sdb = 1;
  pm = 7;
  mlm(x,n,pm,fs,s,freq,len,sdb);
  fp = fopen("mlm1.dat","w");
  for (i=0;i<len;i++)
    { fprintf(fp,"%lf      %lf\n",freq[i],s[i]); }
  fclose(fp);
}
```

运行结果：

宽带 AR(4)模型的功率谱如图 3－2－7 所示。

例 2：AR(4)模型如下

$$x(n) - 2.76x(n-1) + 3.809x(n-2) - 2.654x(n-3) + 0.924x(n-4) = w(n)$$

其中 $w(n)$是零均值、单位方差的白噪声。这是一个窄带随机过程。用最大似然方法进行功率谱估计。

主函数程序(文件名：mlm2.m)：

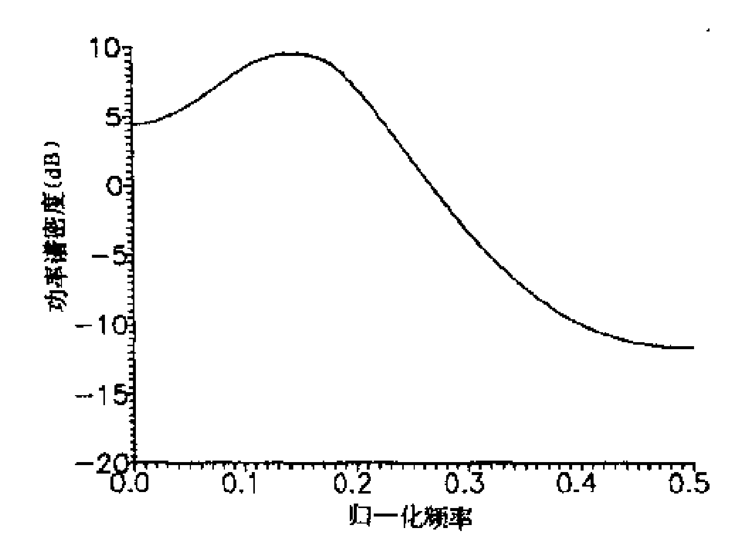


图 3—2—7　宽带 AR(4)模型的功率谱

```
#include "stdio.h"
#include "arma.c"
#include "mlm.c"
main()
{ int i,n,p,q,pm,len,sdb;
  long seed;
  double fs,mean,var;
  static double x[500],freq[100],s[100];
  static double a[5] = {1.0, -2.76, 3.809, -2.654, 0.924};
  static double b[1] = {1.0};
  FILE *fp;
  p = 4;
  q = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 500;
  arma(a,b,p,q,mean,var,&seed,x,n);
  for (i=0;i<300;i++)
    { x[i] = x[i+200]; }
  n = 300;
  len = 100;
  fs = 1.0;
  sdb = 1;
```

```
    pm = 10;
    mlm(x,n,pm,fs,s,freq,len,sdb);
    fp = fopen("mlm2.dat","w");
    for (i=0;i<len;i++)
      { fprintf(fp,"%lf      %lf\n",freq[i],s[i]); }
    fclose(fp);
  }
```

运行结果：

窄带 AR(4)模型的功率谱如图 3－2－8 所示。

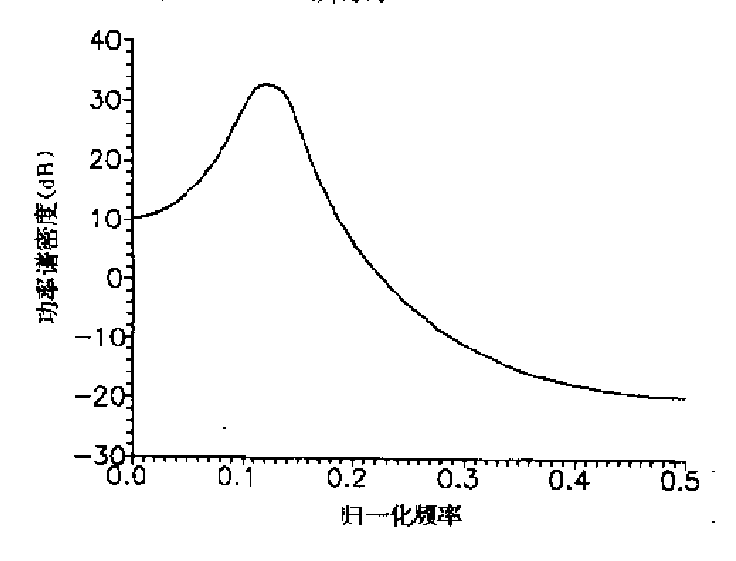


图 3－2－8　窄带 AR(4)模型的功率谱

第三章　时-频分析

§3.1　维格纳(Wigner)分布

一、功　能

用解析信号和快速傅立叶变换计算维格纳(Wigner)分布。

二、方法简介

设连续信号 $s(t)$ 的解析信号为 $x(t)$，则 $s(t)$ 的 Wigner 分布定义为

$$W(t,f) = \int_{-\infty}^{\infty} x\left(t+\frac{\tau}{2}\right) x^*\left(t-\frac{\tau}{2}\right) e^{-j2\pi f\tau} d\tau$$

序列 $s(n)(n=0,1,\cdots,N-1)$ 的离散解析信号为 $x(n)$，则 $s(n)$ 的离散 Wigner 分布定义为

$$W(n,k) = 2\sum_{m=-(M-1)/2}^{(M-1)/2} x(n+m)\, x^*(n-m) e^{-j4\pi mk/M}$$

其中 M 是窗口长度。

离散 Wigner 分布的计算方法如下：

1. 解析信号 $x(n)$ 的计算

(1) 计算序列 $s(n)$ 的 N 点 FFT，得到 $S(k)(k=0,1,\cdots,N-1)$；

(2) 构造 X(k)

$$X(k) = \begin{cases} S(k)\quad, & k=0 \\ 2S(k), & k=1,2,\cdots,\frac{N}{2}-1 \\ 0\quad, & \text{其它} \end{cases}$$

(3) 计算 $X(k)$ 的快速傅立叶反变换，得到解析信号 $x(n)$。

2. 计算下式

$$c(n,m) = \begin{cases} 2x(n+m)\, x^*(n-m)\quad, & 0 \leqslant m \leqslant \frac{M}{2} \\ 2x(n+m-M)\, x^*(n-m+M)\quad, & \frac{M}{2} < m \leqslant M-1 \end{cases}$$

3. 对变量 m，计算 $c(n,m)$ 的快速傅立叶变换(FFT)，得到第 n 时刻的 Wigner 分布 $W(n,k)$

4. 移至下一个窗口，重复步骤 2 和 3，直至计算出所有时刻的 Wigner 分布。

三、使用说明

1. 子函数语句

```
void wigner(x,n,fs,n1,n2,m,fname)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。存放输入序列。

n ——整型变量。输入序列的长度。

fs ——双精度实型变量。信号的采样频率。

n1 ——整型变量。Wigner 分布的起始时刻，n1≥m。

n2 ——整型变量。Wigner 分布的终止时刻，n2≤n－m。

m ——整型变量。Wigner 分布的窗口长度，必须是 2 的整数次幂。

fname ——字符型指针变量，它指向 Wigner 分布的数据文件名 。

四、子函数程序(文件名:wigner.c)

```
#include "stdio.h"
#include "stdlib.h"
#include "analytic.c"
void wigner(x,n,fs,n1,n2,m,fname)
int m,n,n1,n2;
char fname[];
double fs,x[];
{ int i,j;
  double am,freqy, *y, *sr, *si;
  FILE *fp;
  void fft(), analytic();
  sr = malloc(m * sizeof(double));
  si = malloc(m * sizeof(double));
  y = malloc(n * sizeof(double));
  analytic(x,y,n);
  fp = fopen(fname,"w");
  for (i=n1;i<n2;i++)
    { for (j=0;j<m/2;j++)
        { sr[j] = x[i+j] * x[i-j] + y[i+j] * y[i-j];
          si[j] = y[i+j] * x[i-j] - x[i+j] * y[i-j];
        }
      for (j=m/2;j<m;j++)
        { sr[j] = x[i+j-m] * x[i-j+m] + y[i+j-m] * y[i-j+m];
          si[j] = y[i+j-m] * x[i-j+m] - x[i+j-m] * y[i-j+m];
```

```
            }
        fft(sr,si,m,1);
        for (j=0;j<m;j++)
          { am = 2*sqrt(sr[j]*sr[j] +si[j]*si[j]);
            freqy = j*fs/(2.0*m);
            fprintf(fp,"%d %lf %lf\n",i,freqy,am);
          }
      }
    fclose(fp);
    free(sr);
    free(si);
    free(y);
}
```

五、例　题

例 1：线性调频信号 $x(i)$ 为

$$x(i) = \cos[2\pi i\ (f_1 + 0.35\ i)/f_s]$$

选取参数 $f_1=1$，$f_s=210$，n=256，n1 = 40，n2 = 102，m = 32，数据文件名为 wigner1.dat。计算其 Wigner 分布。

主函数程序(文件名:wigner1.m)：

```
#include "stdio.h"
#include "math.h"
#include "wigner.c"
main()
{ int i,n,m,n1,n2;
  double f1,fs,pi;
  static double x[256];
  char fname[]="wigner1.dat";
  void wigner();
  pi = 4.0*atan(1.0);
  f1 = 1;
  fs = 210;
  n = 256;
  for (i=0;i<n;i++)
    { x[i] = cos(2*pi*i*(f1+i*0.35)/fs); }
  m = 32;
  n1 = 40;
  n2 = 102;
```

```
    wigner(x,n,fs,n1,n2,m,fname);
  }
```

运行结果：

线性调频信号 $x(i)$的 Wigner 分布如图 3－3－1 所示。

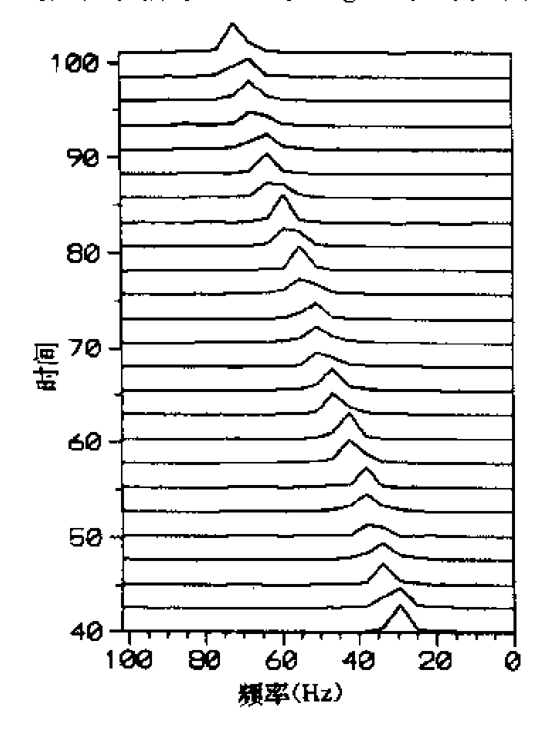


图 3－3－1　线性调频信号的 Wigner 分布

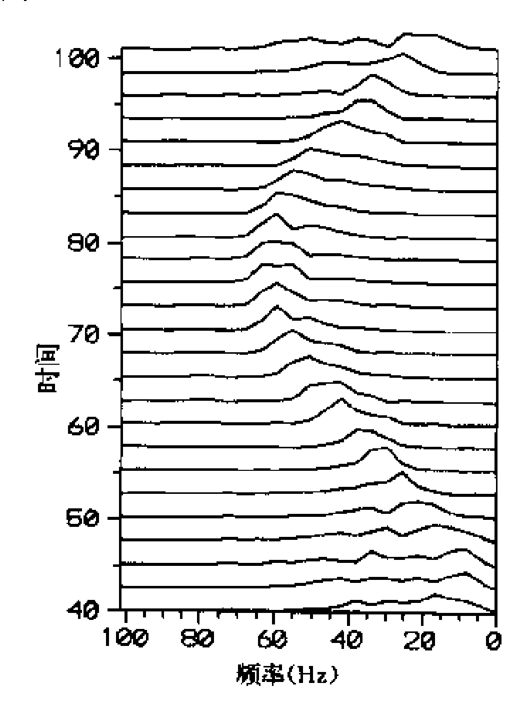


图 3－3－2　调频信号的 Wigner 分布

例 2：调频信号 $x(i)$为

$$x(i) = \cos[2\pi i\ (f_c + 48\cos(2\pi i f_m/f_s))/f_s]$$

选取参数 $f_c=50$，$f_m=1$，$f_s=210$，n=256，n1=40，n2=102，m=32，数据文件名为 wigner2.dat。计算其 Wigner 分布。

主函数程序(文件名：wigner2.m)：

```
#include "stdio.h"
#include "math.h"
#include "wigner.c"
main()
{ int i,n,m,n1,n2;
  double fc,fm,fs,pi;
  static double x[256];
  char fname[]="wigner2.dat";
  void wigner();
  pi = 4.0 * atan(1.0);
  fs = 210.0;
  fc = 50.0;
```

```
    fm = 1.0;
    n = 256;
    for (i=0;i<n;i++)
      { x[i] = cos(2 * pi * i/fs * (fc+48 * cos(2 * pi * i * fm/fs))); }
    m = 32;
    n1 = 40;
    n2 = 102;
    wigner(x,n,fs,n1,n2,m,fname);
  }
```

运行结果：

调频信号 $x(i)$的 Wigner 分布如图 3－3－2 所示。

§3.2 离散小波变换

一、功 能

用 Mallat 算法快速计算一维离散小波变换。

二、方法简介

小波变换是把信号 $x(t)$表示为一簇函数的加权和，而这簇函数是由基本小波 $h(t)$经过伸缩和平移而形成的。伸缩尺度为 a、时间平移为 b 的小波 $h_{a,b}(t)$定义为

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{|a|}} \Psi\left(\frac{t-b}{a}\right) \quad , \quad a \neq 0$$

连续小波变换 $W_x(a,b)$定义为

$$W_x(a,b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} x(t)\, \Psi\left(\frac{t-b}{a}\right) dt$$

对参数 a 和 b 进行离散化，选取 $a=a_0^j$，$b=kb_0a_0^j(a_0>1,b_0>0)$，那么离散小波为

$$\Psi_{j,k}(t) = a_0^{-\frac{j}{2}} \Psi(a_0^{-j} t - kb_0)$$

于是，离散小波变换定义为

$$d_{j,k} = \int_{-\infty}^{\infty} x(t)\, \Psi_{j,k}(t)\, dt$$

若 $a_0=2$，$b_0=1$，那么离散小波变换则称为二进小波变换。此时，二进小波为

$$\Psi_{j,k}(t) = 2^{-\frac{j}{2}} \Psi(2^{-j} t - k)$$

二进小波变换定义为

$$d_{j,k} = \int_{-\infty}^{\infty} x(t)\, \Psi_{j,k}(t)\, dt$$

其反变换为

$$x(t) = \sum_j \sum_k d_{j,k}\, \Psi_{j,k}(t)$$

二进小波变换的 Mallat 算法如下：

正变换算法：

$$\begin{cases} c_{j,k} = \sum_n h_{n-2k} c_{j-1,n} \\ d_{j,k} = \sum_n g_{n-2k} c_{j-1,n} \end{cases}, j = 1,2,\cdots,J$$

其中 $c_{j,k}$是原信号的平滑信号，$d_{j,k}$是细节信号，g_n 是与小波函数相关的带通滤波器的脉冲响应，h_n 是与尺度函数相关的低通滤波器的脉冲响应。

反变换（重构）算法：

$$c_{j-1,k} = \sum_n [h_{k-2n} c_{j,n} + g_{k-2n} d_{j,n}], j = J, J-1, \cdots, 1$$

三、使用说明

1. 子函数语句

正变换：

```
void dwt(g,h,wlen,c,d,m,sca)
```

反变换：

```
void idwt(g,h,wlen,c,d,m,sca)
```

2. 形参说明

g ——双精度实型一维数组，长度为 wlen。尺度系数。

h ——双精度实型一维数组，长度为 wlen。小波系数。

wlen ——整型变量。小波长度。

c ——双精度实型一维数组，长度为$\left(2n-\frac{n}{2^J}\right)$，其中 n 是输入信号的长度。对于 dwt()：0～(sca[0]−1)是输入信号，$\sum_{i=0}^{k-1}sca[i]\sim\left[\sum_{i=0}^{k}sca[i]\right]-1$ 是小波分解的第 k 级平滑信号；对于 idwt()：开始时存放原信号和小波分解的各级平滑信号，最后在 0～(n−1)内存放重建信号。

d ——双精度实型一维数组，长度为$\left(2n-\frac{n}{2^J}\right)$。$\sum_{i=0}^{k-1}sca[i]\sim\left[\sum_{i=0}^{k}sca[i]\right]-1$ 是小波分解的第 k 级细节信号。

m ——整型变量。小波分解的级数。

sca ——双精度实型一维数组，长度为(m+1)。存放小波分解时每级的数据长度。sca[0]是原信号的长度，sca[i]是小波分解时第 i 级的数据长度。

四、子函数程序(文件名：dwt.c 和 idwt.c)

正变换子函数(dwt.c)：

```
void dwt(g,h,wlen,c,d,m,sca)
int m,wlen,sca[];
double c[],d[],g[],h[];
```

```
{ int i,j,k,mid,flag[20];
  double p,q;
  for (flag[0]=0,i=0;i<m;i++)
    { flag[i+1] = flag[i]-sca[i]; }
  printf("\n Doing decomposition:");
  for (j=1;j<=m;j++)
    { printf(" * * * ");
      for (i=0;i<sca[j];i++)
        { p = 0;
          q = 0;
          for (k=0;k<wlen;k++)
            { mid = k + 2*i;
              if (mid >= sca[j-1]) mid = mid - sca[j-1];
              p = p + h[k]*c[flag[j-1]+mid];
              q = q + g[k]*c[flag[j-1]+mid];
            }
          c[flag[j]+i] = p;
          d[flag[j]+i] = q;
        }
    }
}
```

反变换子函数(idwt.c):

```
void idwt(g,h,wlen,c,d,m,sca)
int m,wlen,sca[];
double c[],d[],g[],h[];
{ int i,j,k,mid,flag[20];
  double p,q;
  for (flag[0]=0,j=0;j<m;j++)
    { flag[j+1] = flag[j] + sca[j]; }
  printf("\n Doing reconstruction:");
  for (k=m;k>0;k--)
    { printf(" * * * ");
      for (i=0;i<sca[k];i++)
        { p = 0;
          q = 0;
          for (j=0;j<wlen/2;j++)
            { mid = i - j;
```

```
                if (mid < 0) mid = sca[k]+(i-j);
                p+=h[2*j]*c[flag[k]+mid]+g[2*j]*d[flag[k]+mid];
            q+=h[2*j+1]*c[flag[k]+mid]+g[2*j+1]*d[flag[k]+mid];
              }
            c[flag[k-1]+2*i] = p;
            c[flag[k-1]+2*i+1] = q;
          }
      }
  }
```

五、例　题

例 1：信号 $x(i)$ 为

$$x(i)=\begin{cases}\sin(2\pi i f_1/f_s), & 0\leqslant i<n/2\\ \sin(2\pi i f_2/f_s), & n/2\leqslant i\leqslant n-1\end{cases}$$

选取参数 $f_1=5$，$f_2=10$，$f_s=320$，n = 512，分解 6 级，用 3 阶 Daubechies 小波对该信号进行小波变换。

主函数程序（文件名:dwt1. m）：

```
#include "stdio.h"
#include "math.h"
#include "dwt.c"
#include "idwt.c"
main()
{ int i,j,m,n,wlen, sca[20],flag[20];
  double f1,f2,fs,pi,freq;
  static double c[2000],d[2000];
  static double h[]={ 0.332670552950, 0.806891509311, 0.459877502118,
                     -0.135011020010,-0.085441273882, 0.035226291882};
  static double g[]={ 0.0352263,     0.08544127, -0.135011,
                     -0.459877502118,   0.8068915,   -0.33267055};
  char d—name[8]="d0.dat", c—name[8]="c0.dat";
  FILE *fpc, *fpd;
  void dwt(), idwt();
  printf("\ninput the number of decomposing scales\n");
  scanf(" %d",&m);
  printf("\ninput the lenght of the data(<1000)\n");
  scanf(" %d",&n);
  pi = 4.0*atan(1.0);
  f1 = 5.0;
```

```
f2 = 10.0;
fs = 320.0;
for (i=0;i<n/2;i++)
  { c[i] = sin(2*pi*i*f1/fs); }
for (i=n/2;i<n;i++)
  { c[i] = sin(2*pi*i*f2/fs); }
  fpd = fopen("dwt1.dat","w");
for (i=0;i<n;i++)
  { freq = i*fs/(2*(n-1));
    fprintf(fpd,"%lf      %lf\n",freq,c[i]);
  }
fclose(fpd);
j = n;
flag[0] = 0;
for (i=0;i<=m;i++)
  { flag[i+1] = flag[i] + j;
    sca[i] = j;
    j = j/2;
  }
wlen = 6;
dwt(g,h,wlen,c,d,m,sca);
for (i=1;i<=m;i++)
  { d-name[1]++;
    c-name[1]++;
    fpd = fopen(d-name,"w");
    fpc = fopen(c-name,"w");
    for (j=0;j<sca[i];j++)
      { freq = j*fs/(2*(sca[i]-1));
        fprintf(fpc,"%lf    %lf\n",freq,c[flag[i]+j]);
        fprintf(fpd,"%lf    %lf\n",freq,d[flag[i]+j]);
      }
    fclose(fpd);
    fclose(fpc);
  }
for (i=0;i<sca[m];i++)
        { c[flag[m]+i] = c[flag[m]+i]; }
idwt(g,h,wlen,c,d,m,sca);
  fpd = fopen("idwt1.dat","w");
```

```
    for (i=0;i<n;i++)
      { freq = i*fs/(2*(n-1));
              fprintf(fpd,"%lf        %lf\n",freq,c[i]);
      }
}
```

运行结果：

原始信号和重构信号分别如图 3—3—3 和图 3—3—4 所示。

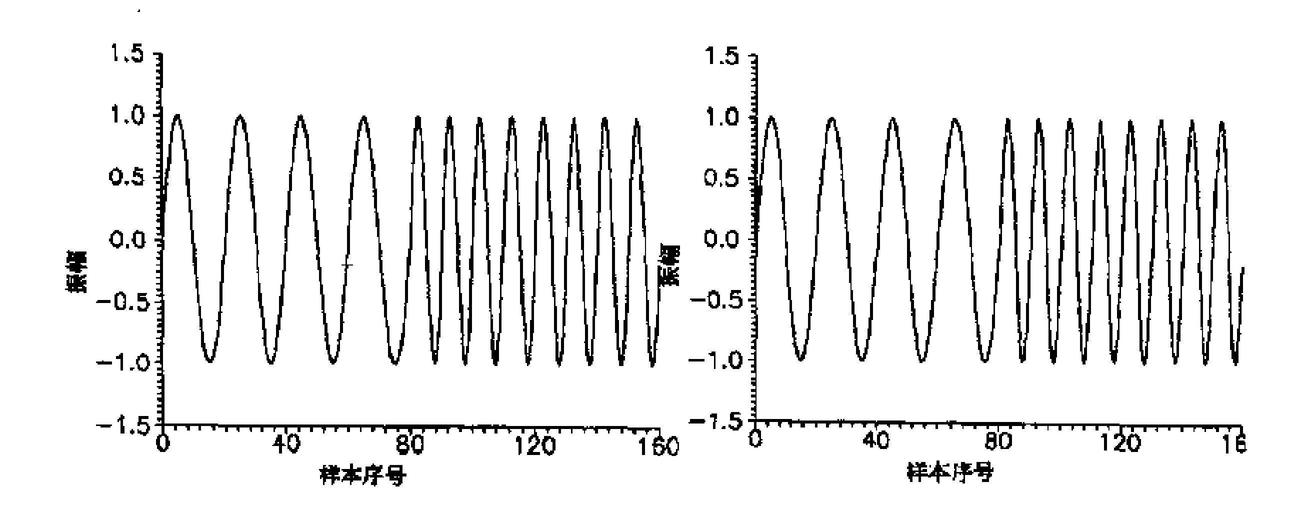


图 3—3—3　小波变换的原始信号　　图 3—3—4　小波变换的重构信号

6 级分解后的平滑信号 $c_i(i=1,2,\cdots,6)$ 如图 3—3—5a、图 3—3—5b、图 3—3—5c、图 3—3—5d、图 3—3—5e、图 3—3—5f 所示，6 级分解后的细节信号 $d_i(i=1,2,\cdots,6)$ 如图 3—3—6a、图 3—3—6b、图 3—3—6c、图 3—3—6d、图 3—3—6e、图 3—3—6f 所示。

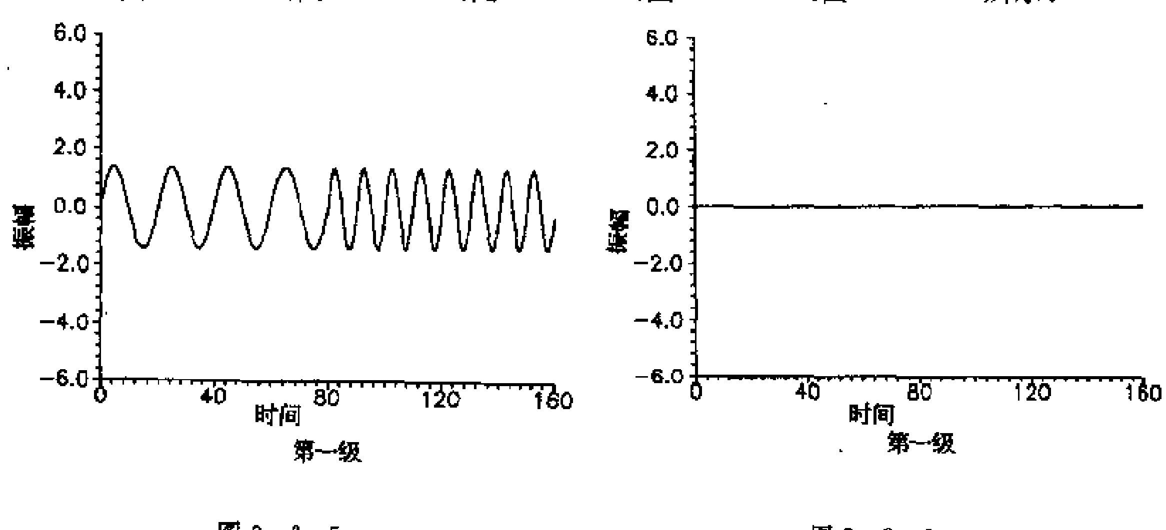


图 3—3—5a　　图 3—3—6a

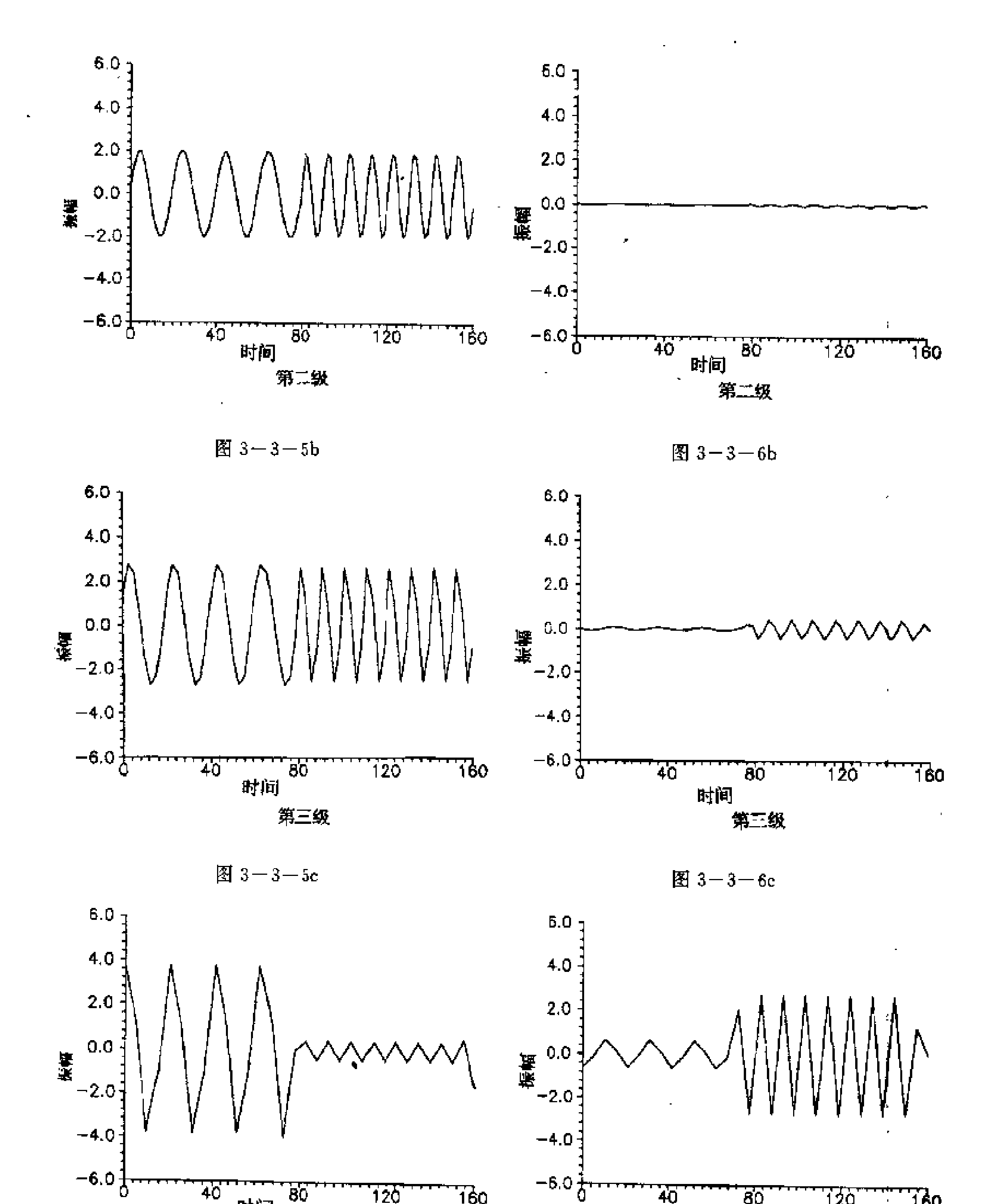


图 3—3—5b

图 3—3—6b

图 3—3—5c

图 3—3—6c

图 3—3—5d

图 3—3—6d

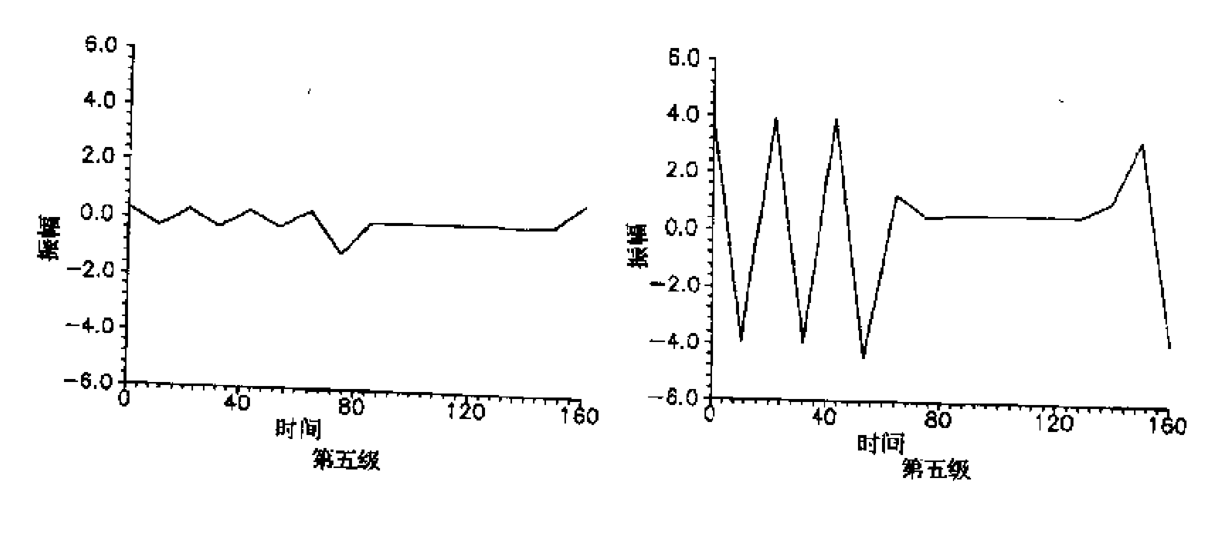


图 3—3—5e

图 3—3—6e

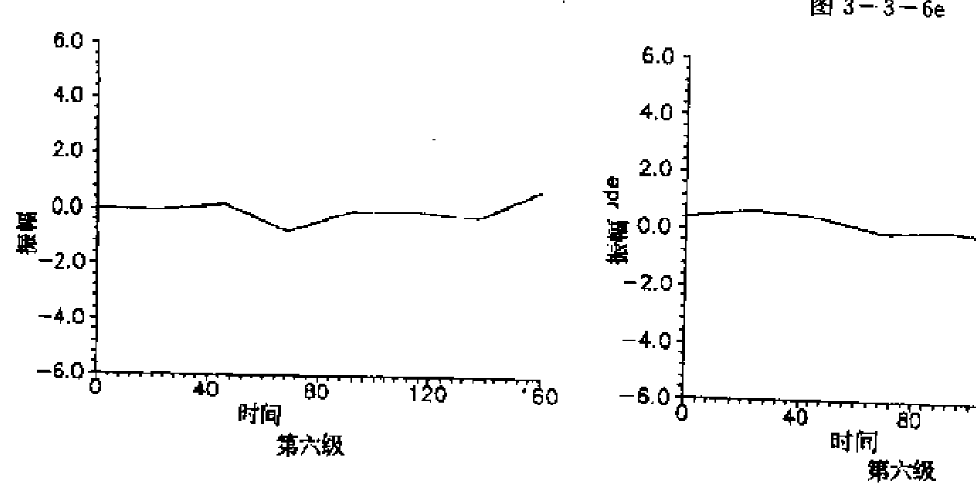


图 3—3—5f

图 3—3—6f

例 2：信号 $x(i)$ 为

$$x(i)=\begin{cases}0, & 0\leqslant i<n/4\\ \sin(2\pi i f/f_s), & n/4\leqslant i<n/2\\ 0, & n/2\leqslant i\leqslant n-1\end{cases}$$

选取参数 $f=10$，$f_s=320$，$n=512$，分解 6 级，用 3 阶 Daubechies 小波对该信号进行小波变换。

主函数程序（文件名：dwt2. m）：

```
#include "stdio.h"
#include "math.h"
#include "dwt.c"
#include "idwt.c"
main()
{ int i,j,m,n,wlen, sca[20],flag[20];
  double f,fs,pi,freq;
```

```
static double c[2000],d[2000];
static double h[]={ 0.332670552950, 0.806891509311, 0.459877502118,
                    -0.135011020010,-0.085441273882, 0.035226291882};
static double g[]={ 0.0352263,     0.08544127, -0.135011,
                    -0.459877502118,   0.8068915, -0.33267055};
char d-name[8]="d0.dat", c-name[8]="c0.dat";
FILE *fpc, *fpd;
void dwt(), idwt();
printf("\ninput the number of decomposing scales\n");
scanf("%d",&m);
printf("\ninput the lenght of the data(<1000)\n");
scanf("%d",&n);
pi = 4.0*atan(1.0);
f = 10.0;
fs = 320.0;
for (i=0;i<n/4;i++)
  { c[i] = 0.0; }
for (i=n/4;i<n/2;i++)
  { c[i] = sin(2*pi*i*f/fs); }
for (i=n/2;i<n;i++)
  { c[i] = 0.0; }
fpd = fopen("dwt2.dat","w");
for (i=0;i<n;i++)
  { freq = i*fs/(2*(n-1));
    fprintf(fpd,"%lf    %lf\n",freq,c[i]);
  }
fclose(fpd);
j = n;
flag[0] = 0;
for (i=0;i<=m;i++)
  { flag[i+1] = flag[i] + j;
    sca[i] = j;
    j = j/2;
  }
wlen = 6;
dwt(g,h,wlen,c,d,m,sca);
for (i=1;i<=m;i++)
  { d-name[1]++;
```

```
        c-name[1]++;
        fpd = fopen(d-name,"w");
        fpc = fopen(c-name,"w");
        for (j=0;j<sca[i];j++)
          { freq = j * fs/(2 * (sca[i]-1));
            fprintf(fpc," %lf    %lf\n",freq,c[flag[i]+j]);
            fprintf(fpd," %lf    %lf\n",freq,d[flag[i]+j]);
          }
        fclose(fpd);
        fclose(fpc);
      }
    for (i=0;i<sca[m];i++)
      { c[flag[m]+i] = c[flag[m]+i]; }
    idwt(g,h,wlen,c,d,m,sca);
    fpd = fopen("idwt2.dat","w");
    for (i=0;i<n;i++)
      { freq = i * fs/(2 * (n-1));
        fprintf(fpd," %lf    %lf\n",freq,c[i]);
      }
  }
```

运行结果：

原始信号和重构信号分别如图 3－3－7 和图 3－3－8 所示。

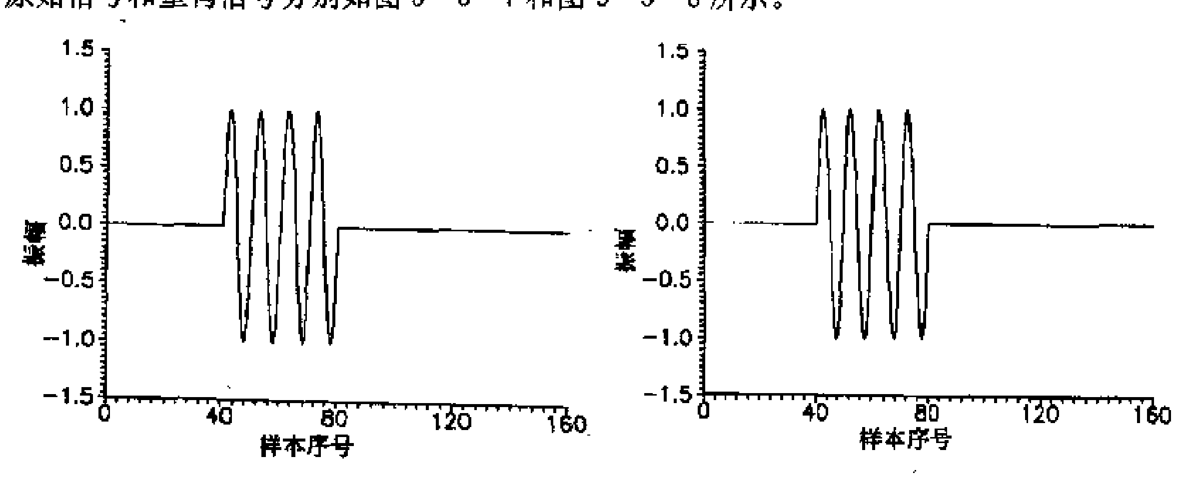
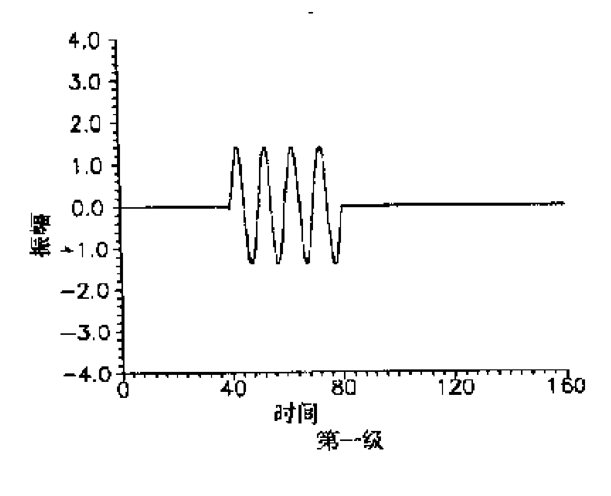
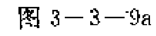


图 3－3－7　小波变换的原始信号　　　图 3－3－8　小波变换的重构信号

6 级分解后的平滑信号 $c_i(i=1,2,\cdots,6)$如图 3－3－9a、图 3－3－9b、图 3－3－9c、图 3－3－9d、图 3－3－9e、图 3－3－9f 所示，6 级分解后的细节信号 $d_i(i=1,2,\cdots,6)$如图 3－3－10a、图 3－3－10b、图 3－3－10c、图 3－3－10d、图 3－3－10e、图 3－3－10f 所示。




图 3—3—9a

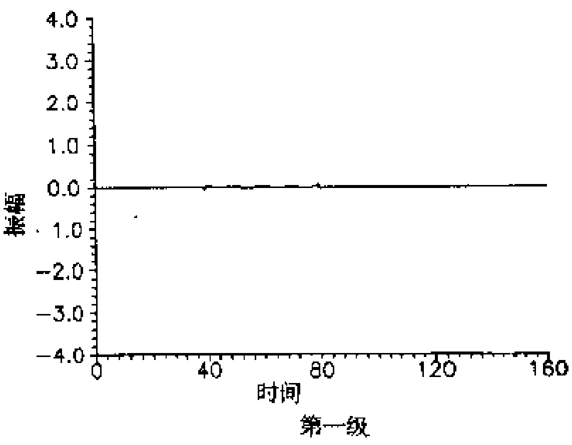


图 3—3—10a

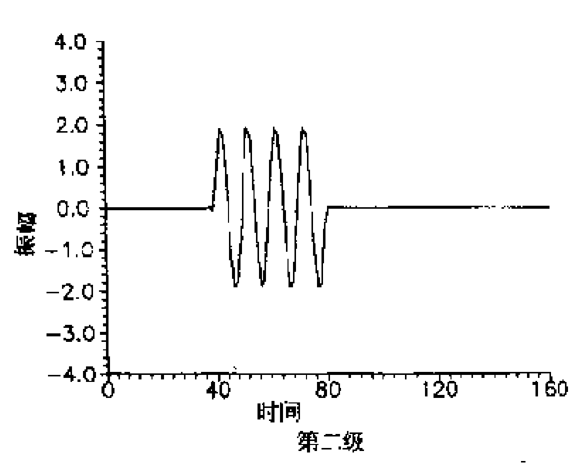


图 3—3—9b

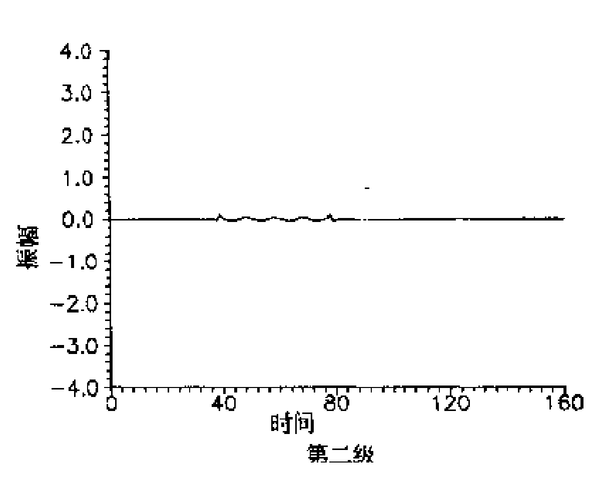


图 3—3—10b

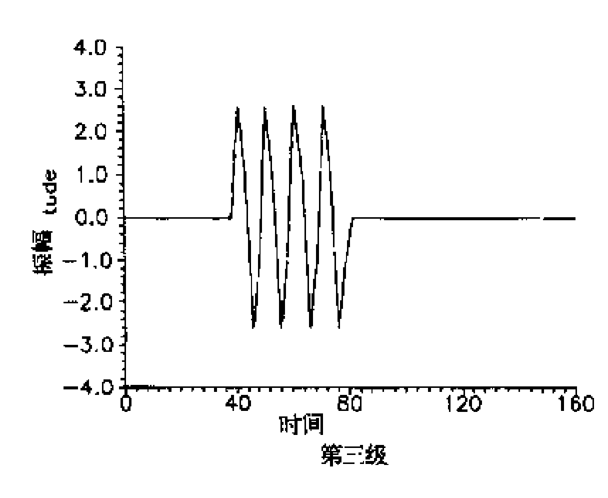


图 3—3—9c

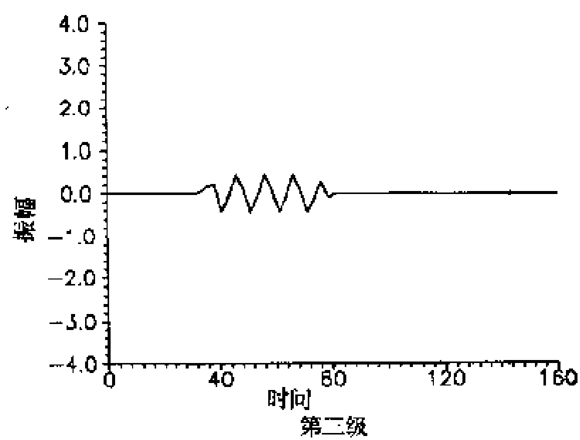


图 3—3—10c

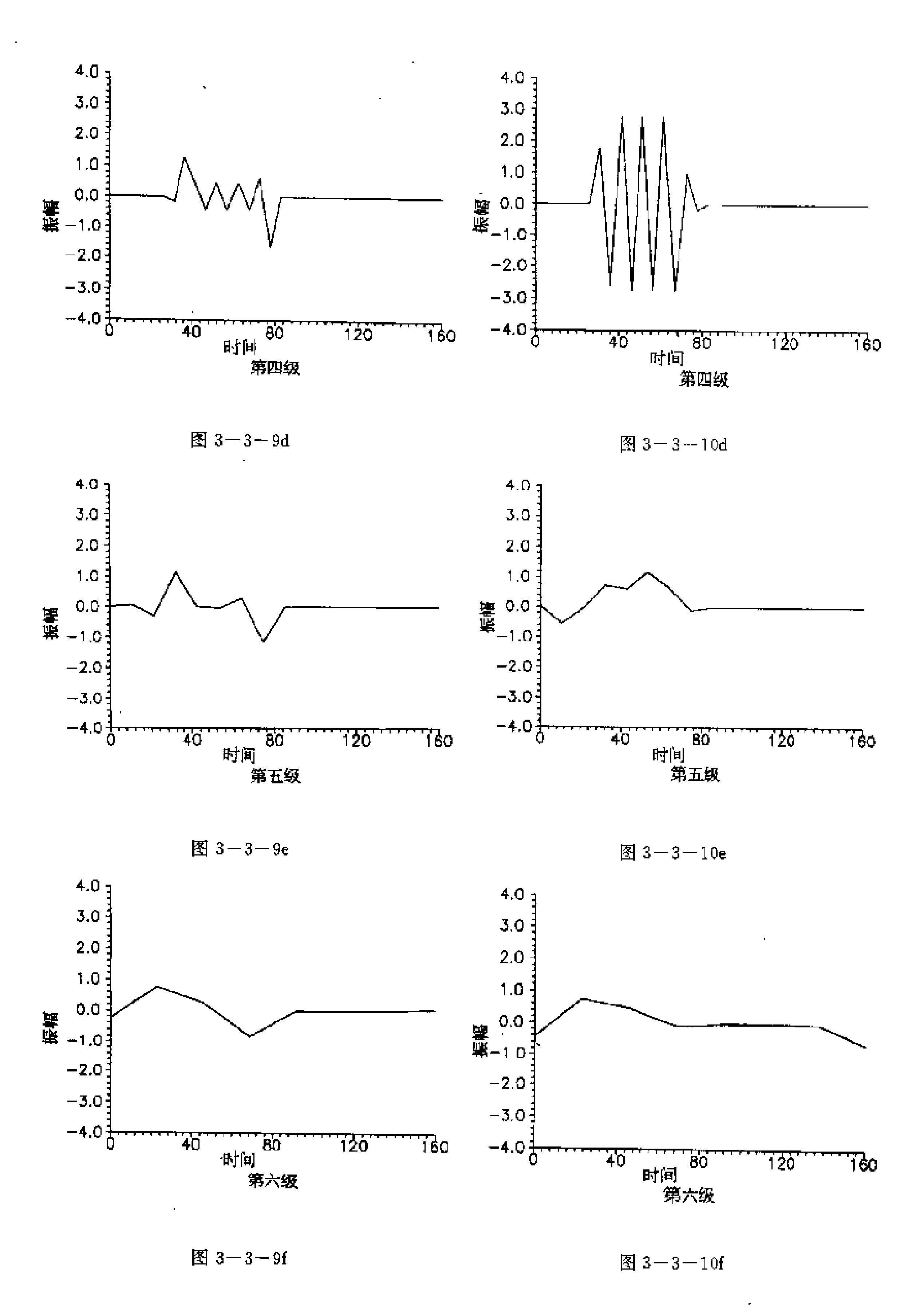


图 3—3—9d

图 3—3—10d

图 3—3—9e

图 3—3—10e

图 3—3—9f

图 3—3—10f

第四章　随机信号的数字滤波

§4.1　维纳(Wiener)数字滤波

一、功　能

对随机信号进行维纳数字滤波。

二、方法简介

设平稳随机序列 $x(n)$ 由信号 $s(n)$ 和噪声 $v(n)$ 迭加而成

$$x(n) = s(n) + v(n)$$

我们希望利用以往所有的 $x(n)$ 值来估计当前的目标信号 $d(n)$。通常，$d(n)$ 具有如下形式

$$d(n) = s(n + m)$$

若 $m = 0$，则是滤波问题；若 $m \leqslant -1$，则是平滑问题；若 $m \geqslant 1$，则是预测问题。

根据维纳滤波理论，可以设计一个 p 阶 FIR 数字滤波器 $h(n)(n=0,1,\cdots,p)$，用它对 $x(n)$ 进行滤波

$$y(n) = \sum_{i=0}^{p} h(i)x(n-i)$$

得到的输出 $y(n)$ 就是目标信号 $d(n)$ 的最小均方误差估计。

均方误差定义为

$$E\{\varepsilon^2(n)\} = E\{[d(n) - y(n)]^2\}$$

为使均方误差最小，可利用正交投影原理，从而得到维纳-霍夫(Wiener-Hopf)方程

$$\sum_{i=0}^{p} r_{xx}(j-i)h(i) = r_{dx}(j) \quad , \quad j = 0,1,\cdots,p$$

即

$$\begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \cdots & r_{xx}(p) \\ r_{xx}(1) & r_{xx}(0) & \cdots & r_{xx}(p-1) \\ \vdots & \vdots & \ddots & \vdots \\ r_{xx}(p) & r_{xx}(p-1) & \cdots & r_{xx}(0) \end{bmatrix} \begin{bmatrix} h(0) \\ h(1) \\ \vdots \\ h(p) \end{bmatrix} = \begin{bmatrix} r_{dx}(0) \\ r_{dx}(1) \\ \vdots \\ r_{dx}(p) \end{bmatrix}$$

最小均方误差为

$$\min E\{\varepsilon^2(n)\} = r_{dd}(0) - \sum_{i=0}^{p} r_{dx}(i)h(i)$$

其中 $r_{dx}(i)=E\{d(n+i)x(n)\}$ 是 $d(n)$ 与 $x(n)$ 的互相关函数，$r_{xx}(i)=E\{x(n+i)x(n)\}$ 是 $x(n)$ 的自相关函数，$r_{dd}(0) = E\{d^2(n)\}$。

维纳-霍夫方程具有托布利兹形式，因而可用本篇§2.1节的莱文森算法有效地求解。

三、使用说明

1. 子函数语句

void wiener(rxx,rdx,p,h,e,x,y,n)

2. 形参说明

rxx ——双精度实型一维数组，长度为(p+1)。信号 $x(n)$的自相关函数 $r_{xx}(i)$。

rdx ——双精度实型一维数组，长度为(p+1)。信号 $d(n)$与 $x(n)$的互相关函数 $r_{dx}(i)$。

p ——整型变量。维纳滤波器的阶数。

h ——双精度实型一维数组，长度为(p+1)。维纳滤波器的系数。

e ——双精度实型变量。维纳滤波器的最小均方误差。

x —— 双精度实型一维数组，长度为 n。存放输入信号 $x(i)$。

y ——双精度实型一维数组，长度为 n。存放维纳滤波后的输出信号 $y(i)$。

n ——整型变量。输入信号的长度。

四、子函数程序(文件名：wiener.c)

```
#include "levin.c"
void wiener(rxx,rdx,p,h,e,x,y,n)
int p,n;
double *e,x[],y[],h[],rxx[],rdx[];
{ int i,k;
  double sum;
  levin(rxx,rdx,p+1,h);
  sum = 0.0;
  for (i=0;i<=p;i++)
    { sum += rdx[i]*h[i]; }
  *e = rdx[0] - sum;
  for (k=0;k<n;k++)
    { y[k] = 0.0;
      for (i=0;i<=p;i++)
        { if ((k-i) >= 0)
            { y[k] += h[i]*x[k-i]; }
        }
    }
}
```

五、例 题

例 1:信号 $s(n)$ 的自相关函数为 $r_{ss}(i)=4(0.5)^{|i|}$,与其正交的白噪声 $v(n)$ 的方差 $\sigma^2=2$,输入信号 $x(n)$ 为 $s(n)$ 与 $v(n)$ 的迭加,这样 $r_{dx}(i)=r_{ss}(i)$,$r_{xx}(i)=r_{ss}(i)+2\delta(i)$,试设计一个 $p=2$ 阶的维纳滤波器。

主函数程序(文件名:wiener1.m):

```
#include "stdio.h"
#include "math.h"
#include "wiener.c"
main()
{ int i,p,n;
  double e,rxx[3],rdx[3],h[3],x[10],y[10];
  n = 1;
  p = 2;
  for (i=0;i<=p;i++)
    { rdx[i] = 4.0*pow(0.5,i); }
  rxx[0] = rdx[0] + 2.0;
  for (i=1;i<=p;i++)
    { rxx[i] = rdx[i]; }
  wiener(rxx,rdx,p,h,&e,x,y,n);
  printf("\nThe Coefficients of Wiener Filter\n");
  for (i=0;i<=p;i++)
    { printf("h(%d) = %10.7lf\n",i,h[i]); }
  printf("The Minimum MSE Error = %lf\n",e);
}
```

运行结果:

维纳滤波器的系数为

```
h(0) =    0.6235294
h(1) =    0.1176471
h(2) =    0.0235294
```

最小均方误差为

The Minimum MSE Error = 1.247059

例 2:设 $s(n)$ 为一个 AR(2)随机过程

$$s(n)-0.1s(n-1)-0.8s(n-2)=v_1(n)$$

其中 $v_1(n)$ 为零均值、单位方差的白噪声。又设 $x(n)$ 为 $s(n)$ 与另一个零均值、单位方差的白噪声 $v(n)$ 的迭加

$$x(n)=s(n)+v(n)$$

其中 $v(n)$ 与 $s(n)$ 相互独立。选取目标信号 $d(n)=s(n)$,这是维纳滤波问题,并有

$$r_{dx}(i) = r_{sx}(i) = r_{xx}(i)$$
$$r_{xx}(i) = r_{ss}(i) + \delta(i)$$

设计维纳滤波器，并对序列 $x(n)$进行滤波。

主函数程序(文件名:wiener2.m):

```
#include "stdio.h"
#include "arma.c"
#include "wiener.c"
main()
{ int i,k,n,p,ip,iq;
  long seed;
  double e,mean,var;
  static double a[3] = {1.0, -0.1, -0.8};
  static double b[1] = {1.0};
  static double rxx[30],rdx[30];
  static double h[30],x[100],y[100],s[100];
  void wiener();
  FILE *fp;
  ip = 2;
  iq = 0;
  seed = 13579l;
  mean = 0.0;
  var = 1.0;
  n = 100;
  arma(a,b,ip,iq,mean,var,&seed,s,n);
  fp = fopen("wieners.dat","w");
  for (i=0;i<n;i++)
    { fprintf(fp,"%d      %lf\n",i,s[i]); }
  fclose(fp);
  seed = 157l;
  for (i=0;i<n;i++)
    { x[i] = s[i] + gauss(0.0,1.0,&seed); }
  fp = fopen("wienerx.dat","w");
  for (i=0;i<n;i++)
    { fprintf(fp,"%d      %lf\n",i,x[i]); }
  fclose(fp);
  p = 8;
  for (k=0;k<=p;k++)
    { rdx[k] = 0.0;
```

```
          for (i=0;i<(n-k);i++)
            { rdx[k] += s[i] * s[i+k]; }
          rdx[k] = rdx[k]/n;
        }
      rxx[0] = rdx[0] + 1.0;
      for (i=1;i<=p;i++)
        { rxx[i] = rdx[i]; }
      wiener(rxx,rdx,p,h,&e,x,y,n);
      printf("\nThe Coefficients of Wiener Filter\n");
      for (i=0;i<=p;i++)
        { printf("h(%d) = %10.7lf\n",i,h[i]); }
      printf("The Minimum MSE Error = %lf\n",e);
      fp = fopen("wienery.dat","w");
      for (i=0;i<n;i++)
        { fprintf(fp,"%d       %lf\n",i,y[i]); }
      fclose(fp);
    }
```

运行结果：

维纳滤波器的系数为

h(0) = 0.5763951
h(1) = 0.0091187
h(2) = 0.2055788
h(3) = 0.0342110
h(4) = 0.0696614
h(5) = 0.0109759
h(6) = 0.0182548
h(7) = 0.0258620
h(8) = 0.0156630

最小均方误差为

The Minimum MSE Error = 0.576395

信号 $s(n)$、迭加噪声的输入信号 $x(n)$ 和滤波后的输出信号 $y(n)$ 分别如图 3—4—1、图 3—4—2 和图 3—4—3 所示。

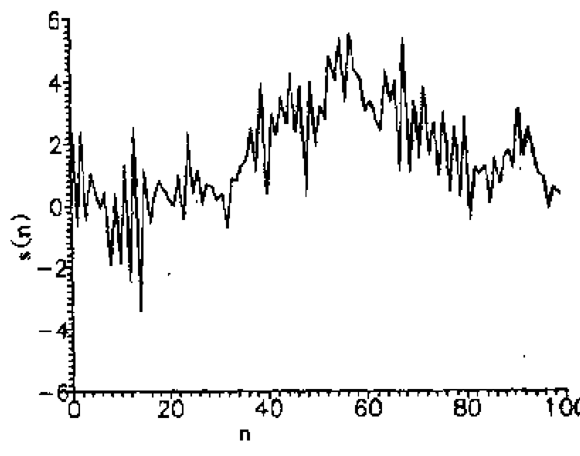


图 3—4—1 信号 $s(n)$

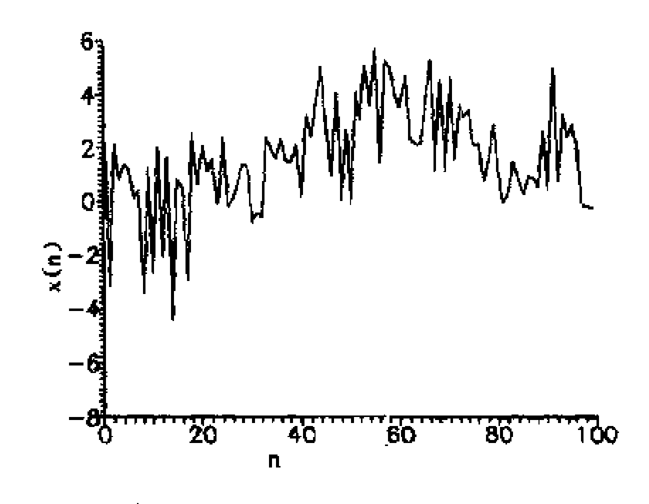


图 3－4－2 迭加噪声的输入信号 $x(n)$

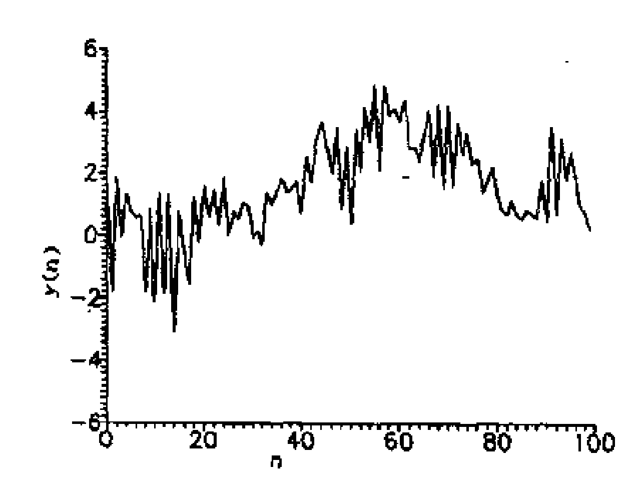


图 3－4－3 维纳滤波后的输出信号 $y(n)$

§4.2 卡尔曼(Kalman)数字滤波

一、功 能

对随机信号进行卡尔曼滤波

二、方法简介

离散线性系统的状态方程和观测方程为

$$X_k = \Phi_{k,k-1}X_{k-1} + D_kU_k + B_kW_k$$

$$Z_k = H_kX_k + V_k$$

其中：X_k 是 n 维列向量，表示第 k 时刻系统的状态；$\Phi_{k,k-1}$是 $n\times n$ 维状态转移矩阵；D_k 是 n 维定常列向量，U_k 是标量，表示系统的输入信号；B_k 是 $n\times m$ 维定常矩阵，W_k 是 m

维列向量，表示动态噪声；Z_k 是标量，表示观测值；H_k 是 n 维行向量，称为观测矩阵；V_k 是标量，表示观测噪声。

动态噪声与观测噪声的统计特性为

$$E[W_k]=0 \quad , \quad E[W_k W_j^T]=Q_k\delta_{kj}$$
$$E[V_k]=0 \quad , \quad E[V_k V_j^T]=R_k\delta_{kj}$$
$$E[W_k V_j^T]=0$$

其中 Q_k 是 $m\times m$ 维的动态噪声 W_k 的协方差阵；R_k 是标量，表示观测噪声 V_k 的协方差。

卡尔曼滤波就是从初值 $\hat{X}_0=E[X_0]$和 P_0(给定)出发，利用上面的状态空间模型和观测值 Z_k，递推地计算出每个时刻的状态估计 X_k。具体算法如下：

$$\hat{X}_{k|k-1}=\Phi_{k,k-1}\hat{X}_{k-1}+D_k U_k$$
$$P_{k|k-1}=\Phi_{k,k-1}P_{k-1}\Phi_{k,k-1}^T+B_k Q_k B_k^T$$
$$K_k=P_{k|k-1}H_k^T(R_k+H_k P_{k|k-1}H_k^T)^{-1}$$
$$\hat{X}_k=\hat{X}_{k|k-1}+K_k(Z_k-H_k\hat{X}_{k|k-1})$$
$$P_k=(1-K_k H_k)P_{k|k-1}$$

其中：K_k 是 n 维列向量，表示卡尔曼增益；$\hat{X}_k$是 n 维列向量，表示第 k 时刻的状态估计；$\hat{X}_{k|k-1}$是 n 维列向量，表示状态的一步预测值；P_k 是 $n\times n$ 维的滤波误差方差阵；$P_{k|k-1}$是 $n\times n$ 维预测误差方差阵。

三、使用说明

1. 子函数语句

void kalman(n,m,len,f,d,u,b,q,h,r,z,x,p,g)

2. 形参说明

n ——整型变量。状态向量的维数。

m ——整型变量。动态噪声的维数。

len ——整型变量。观测序列的长度。

f ——双精度实型二维数组，体积为 n×n。系统状态转移矩阵 $\Phi_{k,k-1}$。

d ——双精度实型一维数组，长度为 n。系统输入信号的系数向量 D_k。

u ——双精度实型变量。系统的输入信号 V_k。

b ——双精度实型二维数组，体积为 n×m。动态噪声的系数矩阵 B_k。

q ——双精度实型二维数组，体积为 m×m。动态噪声的协方差矩阵 Q_k。

h ——双精度实型一维数组，长度为 n。系统的观测矩阵 H_k。

r ——双精度实型变量。观测噪声的协方差 R_k。

z ——双精度实型一维数组，长度为 len。系统的观测值 Z_k。

x ——双精度实型一维数组，长度为 n。系统的状态向量 X_k。

p ——双精度实型二维数组，体积为 n×n。系统的滤波误差方差阵 P_k。

g ——双精度实型一维数组，长度为 n。系统的卡尔曼滤波增益 K_k。

四、子函数程序(文件名:kalman.c)

```
#include "stdlib.h"
void kalman(n,m,len,f,d,u,b,q,h,r,z,x,p,g)
int m,n,len;
double r;
double f[],d[],u[],b[],q[],h[],z[],x[],p[],g[];
{ int i,j,k,k1;
  double y, *p1, *c, *s, *cc, *x1;
  p1 = malloc(n * n * sizeof(double));
  c = malloc(n * n * sizeof(double));
  s = malloc(n * n * sizeof(double));
  x1 = malloc(n * sizeof(double));
  cc = malloc(n * sizeof(double));
  for (k1=0;k1<len;k1++)
    { for (i=0;i<n;i++)
        { y = 0.0;
          for (j=0;j<n;j++)
            { y = y + f[i * n+j] * x[j]; }
          x1[i] = y + d[i] * u[k1];
        }
      for (i=0;i<n;i++)
        { for (j=0;j<n;j++)
            { y = 0.0;
              for (k=0;k<n;k++ )
                { y = y + f[i * n+k] * p[k * n+j]; }
              c[i * n+j] = y;
            }
        }
      for (i=0;i<n;i++)
        { for (j=0;j<n;j++)
            { y = 0.0 ;
              for (k=0;k<n;k++)
                { y = y + c[i * n+k] * f[j * n+k]; }
              p1[i * n+j] = y;
            }
        }
      for (i=0;i<n;i++)
```

```
    { for (j=0;j<m;j++)
        { y = 0.0 ;
          for (k=0;k<m;k++)
            { y = y + b[i*m+k]*q[k*m+j]; }
          s[i*m+j] = y ;
        }
    }
  for (i=0;i<n;i++)
    { for (j=0;j<n;j++)
        { y = 0.0;
          for (k=0;k<m;k++)
            { y = y + s[i*m+k]*b[j*m+k] ; }
          p1[i*n+j] = y + p1[i*n+j];
        }
    }
  for (i=0;i<n;i++)
    { y = 0.0 ;
      for (k=0;k<n;k++)
        { y = y + p1[i*n+k]*h[k]; }
      cc[i] = y ;
    }
  y = 0.0 ;
  for (i=0;i<n;i++)
    { y = y + cc[i]*h[i]; }
  y = r + y ;
  for (i=0;i<n;i++)
    { g[i] = cc[i]/y; }
  for (i=0;i<n;i++)
    { for (j=0;j<n;j++)
        { p[i*n+j] = p1[i*n+j] - g[i]*cc[j]; }
    }
  y = 0.0 ;
  for (i=0;i<n;i++)
    { y = y + h[i]*x1[i]; }
  y = z[k1] - y;
  for (i=0;i<n;i++)
    { x[i] = x1[i] + g[i]*y; }
}
```

```
  free(p1);
  free(c);
  free(s);
  free(x1);
  free(cc);
}
```

五、例　题

二阶线性系统的状态方程和观测方程为

$$X(k) = \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} X(k-1)$$

$$Z(k) = [1,0]\, X(k) + V(k)$$

观测值 $Z(1)=1.1$，$Z(2)=2.0$，$Z(3)=3.2$，$Z(4)=3.8$，观测噪声 $V(k)$ 的方差 R 为 0.1，状态向量和滤波误差方差矩阵的初始值分别为

$$\hat{X}(0) = \begin{bmatrix}0\\0\end{bmatrix} \qquad P(0) = \begin{bmatrix}10 & 0\\0 & 10\end{bmatrix}$$

用卡尔曼滤波器计算状态向量的估计、滤波器增益和滤波误差方差矩阵。此处仅给出前 4 次递推计算的结果。

主函数程序(文件名:kalman.m):

```
#include "math.h"
#include "stdio.h"
#include "kalman.c"
main()
{ int i,j,m,n,len;
  static double f[4] = {1.0, 1.0, 0.0, 1.0};
  static double b[4] = {0.0, 0.0, 0.0, 0.0};
  static double d[2] = {0.0, 0.0};
  static double h[2] = {1.0, 0.0};
  static double q[4] = {0.0, 0.0, 0.0,0.0};
  static double x[2] = {0.0, 0.0};
  static double p[4] = {10.0, 0.0, 0.0, 10.0};
  static double u[150] = {0.0, 0.0, 0.0, 0.0};
  static double z[150] = {1.1, 2.0, 3.2, 3.8};
  double g[2], r=0.1;
  m = 2;
  n = 2;
  len = 4;
  kalman(n,m,len,f,d,u,b,q,h,r,z,x,p,g);
```

```
    printf("The Error Variance Matrix of Kalman Filter\n");
    for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      { printf(" p(%d,%d) = %10.8lf\n",i,j,p[i*n+j]); }
    printf("The Kalman Gain\n");
    for (i=0;i<n;i++)
      { printf(" k(%d) = %10.8lf\n",i,g[i]); }
    printf("The State Estimate of Kalman Filter\n");
    for (i=0;i<n;i++)
      { printf(" x(%d) = %10.8lf\n",i,x[i]); }
}
```

运行结果

第一次迭代：

```
p(0,0) = 0.09950249      p(0,1) = 0.04975124
p(1,0) = 0.04975124      p(1,1) = 5.02487562
k(0)   = 0.99502488      k(1)   = 0.49751244
x(0)   = 1.09452736      x(1)   = 0.54726368
```

第二次迭代：

```
p(0,0) = 0.09812167      p(0,1) = 0.09531819
p(1,0) = 0.09531819      p(1,1) = 0.18783291
k(0)   = 0.98121671      k(1)   = 0.95318195
x(0)   = 1.99327166      x(1)   = 0.88870199
```

第三次迭代：

```
p(0,0) = 0.08265668      p(0,1) = 0.04910779
p(1,0) = 0.04910779      p(1,1) = 0.04878365
k(0)   = 0.82656683      k(1)   = 0.49107794
x(0)   = 3.14484368      x(1)   = 1.04487772
```

第四次迭代：

```
p(0,0) = 0.06966534      p(0,1) = 0.02969504
p(1,0) = 0.02969504      p(1,1) = 0.01971475
k(0)   = 0.69665341      k(1)   = 0.29695036
x(0)   = 3.91822066      x(1)   = 0.92914981
```

§ 4.3 最小均方(LMS)自适应数字滤波

一、功　能

最小均方(LMS)算法的自适应数字滤波器。

二、方法简介

设横向自适应数字滤波器的输入为 $x(n)$，理想输出为 $d(n)$，实际输出为 $y(n)$，滤波器的加权系数为 $w_i(n)(i=0,1,\cdots,M-1)$，那么 LMS 算法如下：

$$y(n)=\sum_{i=0}^{M-1}w_i(n)x(n-i)$$

$$e(n)=d(n)-y(n)$$

$$w_i(n+1)=w_i(n)+2\mu e(n)x(n-i)\quad,\quad i=0,1,\cdots,M-1$$

其中 μ 是收敛因子(学习率)。

三、使用说明

1. 子函数语句

```
void lms(x,d,y,n,w,m,mu)
```

2. 形参说明

x ——双精度实型一维数组，长度为 n。输入信号。

d ——双精度实型一维数组，长度为 n。理想输出信号。

y ——双精度实型一维数组，长度为 n。实际输出信号。

n ——整型变量。输入信号的长度。

w ——双精度实型一维数组，长度为 m。自适应滤波器的加权系数。

m ——整型变量。自适应滤波器的长度(阶数−1)。

mu ——双精度实型变量。收敛因子。

四、子函数程序(文件名:lms.c)

```
void lms(x,d,y,n,w,m,mu)
int m,n;
double mu,d[],x[],y[],w[];
{ int i,k;
  double e;
  for (i=0;i<m;i++)
    { w[i] = 0.0; }
  for (k=0;k<m;k++)
    { y[k] = 0.0;
```

```
            for (i=0;i<=k;i++)
              { y[k] += x[k-i] * w[i]; }
            e = d[k] - y[k];
            for (i=0;i<=k;i++)
              { w[i] += 2.0 * mu * e * x[k-i]; }
          }
        for (k=m;k<n;k++)
          { y[k] = 0.0;
            for (i=0;i<m;i++)
              { y[k] += x[k-i] * w[i]; }
            e = d[k] - y[k];
            for (i=0;i<m;i++)
              { w[i] += 2.0 * mu * e * x[k-i]; }
          }
      }
```

五、例　题

用自适应数字滤波器进行谱线增强，输入信号 $x(i)$与理想输出信号 $d(i)$选取如下

$$d(i)=\sqrt{2}\sin\frac{2\pi i}{20}+v(i)$$

$$x(i)=d(i-1)$$

其中 $v(i)$ 是均值为零、方差为 1 的高斯白噪声，正弦信号的数字频率为 0.05，信噪比 SNR=1，$x(i)$比 $d(i)$延迟一个采样间隔。选取样本数 n=500，滤波器的长度 m=20，收敛因子 μ=0.0005。

主函数程序(文件名:lms.m)：

```
    #include "stdio.h"
    #include "math.h"
    #include "gauss.c"
    #include "lms.c"
    main()
    { int i,m,n;
      long seed;
      double mu,pi,mean,sigma;
      static double d[501],x[501],y[501],w[50];
      FILE *fp;
      pi = 4.0 * atan(1.0);
      mean = 0.0;
      sigma = 1.0;
```

```
    seed = 13579l;
    n = 500;
    for (i=0;i<n;i++)
      { d[i] = sqrt(2.0) * sin(2 * pi * i/20.0);
        d[i] += gauss(mean,sigma,&seed);
      }
    for (i=0;i<(n-1);i++)
      { x[i+1] = d[i]; }
    fp = fopen("lmsd.dat","w");
    for (i=0;i<n;i++)
      { fprintf(fp,"%d %lf\n",i,d[i]); }
    fclose(fp);
    m = 20;
    mu = 0.0005;
    lms(x,d,y,n,w,m,mu);
    printf("\n The Coefficients of Adaptive Filter\n");
    for (i=0;i<m;i+=4)
      { printf("     %10.7f     %10.7f",w[i],w[i+1]);
        printf("     %10.7f     %10.7f",w[i+2],w[i+3]);
        printf("\n");
      }
    fp = fopen("lmsy.dat","w");
    for (i=0;i<n;i++)
      { fprintf(fp,"%d     %lf\n",i,y[i]); }
    fclose(fp);
  }
```

运行结果：

自适应滤波器的系数为

0.0802803	0.0737254	0.0374495	0.0179859
−0.0057956	−0.0396579	−0.0482402	−0.0711269
−0.1174914	−0.0873055	−0.0541619	−0.0689547
−0.0379733	−0.0821936	0.0091304	0.0289901
0.0389192	0.0903648	0.0787547	0.0753598

正弦信号与高斯白噪声的混合信号如图 3—4—4 所示。经过自适应谱线增强后，正弦信号得到加强，信噪比明显提高，处理结果如图 3—4—5 所示。

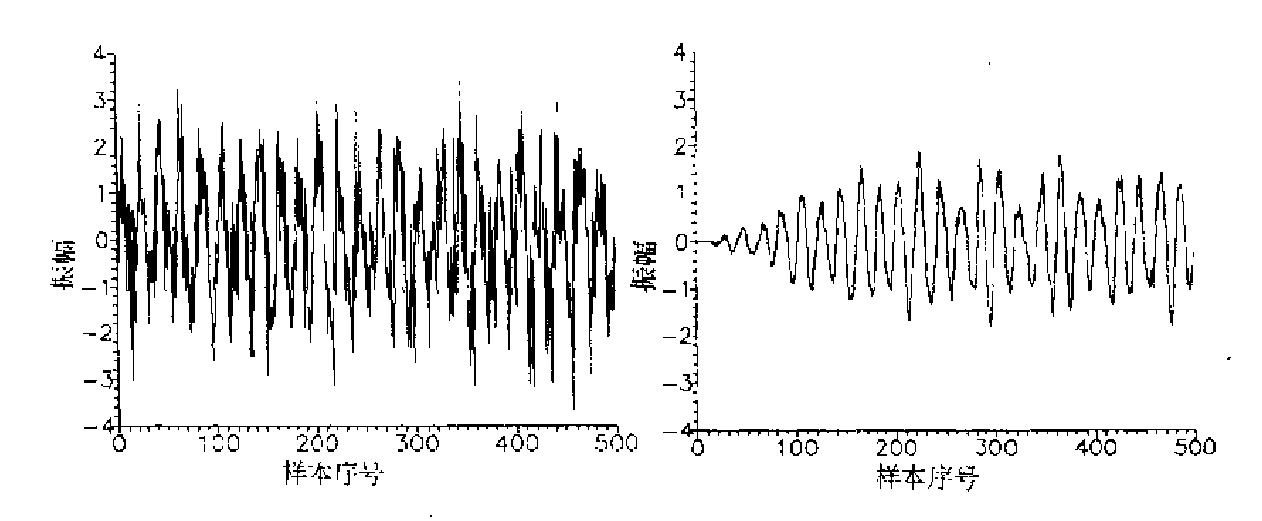


图 3－4－4　受白噪声干扰的正弦信号　　　图 3－4－5　自适应谱线增强后的正弦信号

§4.4　归一化 LMS 自适应数字滤波

一、功　能

归一化最小均方(LMS)算法的自适应数字滤波器。

二、方法简介

设横向自适应数字滤波器的输入为 $x(n)$，理想输出为 $d(n)$，实际输出为 $y(n)$，滤波器的加权系数为 $w_i(n)(i=0,1,\cdots,M-1)$，那么归一化 LMS 算法可描述如下：

$$y(n)=\sum_{i=0}^{M-1} w_i(n)\,x(n-i)$$

$$e(n)=d(n)-y(n)$$

$$w_i(n+1)=w_i(n)+\frac{2\mu}{M\hat{\sigma}^2(n)}e(n)x(n-i)\quad,\quad i=0,1,\cdots,M-1$$

$$\hat{\sigma}^2(n)=\alpha x^2(n)+(1-\alpha)\hat{\sigma}^2(n-1)\quad,\quad 0\leqslant\alpha\ll 1$$

其中 μ 为收敛因子($0<\mu<1$)，α 为遗忘因子，$\hat{\sigma}^2(n)$是输入信号功率的估值。

三、使用说明

1. 子函数语句

void nlms(x,d,n,w,m,mu,sigma2,a,px)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放输入信号，最后存放实际输出信号。

d — 双精度实型一维数组，长度为 n。理想输出信号。

n — 整型变量。输入信号的长度。

w ——双精度实型一维数组，长度为 m。自适应滤波器的加权系数。

m ——整型变量。自适应滤波器的长度(阶数－1)。

mu ——双精度实型变量。学习因子，0<mu<1。

sigma2——双精度实型变量。输入信号的功率估值 $\hat{\sigma}^2$。

a ——双精度实型变量。遗忘因子，0≤a≪1。

px ——双精度实型一维数组，长度为 m。在分块处理时，用于保存输入信号的过去值。

四、子函数程序(文件名:nlms.c)

```
void nlms(x,d,n,w,m,mu,sigma2,a,px)
int m,n;
double a,mu,sigma2,d[],x[],w[],px[];
{ int i,k;
  double e,tmp;
  for (k=0;k<n;k++)
    { px[0] = x[k];
      x[k] = 0.0;
      for (i=0;i<m;i++)
        { x[k] += px[i] * w[i]; }
      e = d[k] - x[k];
      sigma2 = a * px[0] * px[0] + (1.0 - a) * sigma2;
      tmp = 2 * mu/(m * sigma2);
      for (i=0;i<m;i++)
        { w[i] += tmp * e * px[i]; }
      for (i=(m-1);i>=1;i--)
        { px[i] = px[i-1]; }
    }
}
```

五、例　题

用自适应数字滤波器设计通常的 FIR 数字滤波器。具体方法是：在 FIR 滤波器的频带内，选择 L 个频率点 f_1, f_2, …, f_L 。在频率点 $f_i(i=1,2,\cdots,L)$ 处的幅度响应为 A_i ，相移为 θ_i。理想输出信号 $d(k)$ 由 L 个正弦信号组合而成，其中每个正弦信号的频率为 f_i ，振幅为 A_i ，相位为 θ_i，即

$$d(k)=\sum_{i=1}^{L}A_i\sin(2\pi f_i k+\theta_i)$$

输入信号 $x(k)$也由 L 个正弦信号组合而成，但每个正弦信号都是单位幅度、零相移的

$$x(k)=\sum_{i=1}^{L}\sin(2\pi f_i k)$$

于是，自适应滤波器的系数就是满足上述要求的 FIR 数字滤波器的单位冲激响应。选取参数：数据长度 n=500，滤波器的长度 m=51，收敛因子 $\mu=0.2$，遗忘因子 $\alpha=0.0$，输入信号功率初始值设定为 $\hat{\sigma}^2(0)=0.2$，正弦信号数 $L=41$。

主函数程序(文件名:nlms.m)：

```
#include "stdio.h"
#include "math.h"
#include "nlms.c"
#include "gain.c"
main()
{ int i,k,m,n;
  double a,mu,pi,freq,sigma2,px[60];
  static double d[501],x[501],y[201],w[60],amp[60],c[60];
  FILE *fp;
  void nlms(),gain();
  pi = 4.0*atan(1.0);
  for (i=0;i<20;i++)
    { amp[i] = 1.0; }
  amp[20] = 0.5;
  for (i=21;i<=40;i++)
    { amp[i] = 0.0; }
  m = 51;
  n = 500;
  for (k=0;k<n;k++)
    { d[k] = 0.1*amp[0];
      x[k] = 0.1;
      for (i=1;i<41;i++)
        { d[k] += amp[i]*0.1*sin(2.0*pi*0.0125*i*(k-25.0));
          x[k] += 0.1*sin(2.0*pi*0.0125*i*k);
        }
    }
  for (i=0;i<m;i++)
    { w[i] = 0.0; }
  mu = 0.2;
  sigma2 = 0.2;
  a = 0.0;
  for (i=0;i<m;i++)
    { px[i] = 0.0; }
  nlms(x,d,n,w,m,mu,sigma2,a,px);
```

```
    printf("\n The Coefficients of Adaptive Filter\n");
    for (i=0;i<m;i+=4)
      { printf("        %10.7lf        %10.7lf",w[i],w[i+1]);
        if ( (i+2) <= 49 )
          { printf("        %10.7lf        %10.7lf",w[i+2],w[i+3]); }
        else
          { printf("        %10.7lf",w[i+2]); }
        printf("\n");
      }
    fp = fopen("nlmshn.dat","w");
    for (i=0;i<m;i++)
      { fprintf(fp,"%d      %10.7lf\n",i,w[i]); }
    fclose(fp);
    c[1] = 0.0;
    fp = fopen("nlmsam.dat","w");
    gain(w,c,m-1,1,x,y,200,1);
    for (i=0;i<200;i++)
      { freq = 0.5*i/199;
        fprintf(fp,"%lf      %lf \n",freq,x[i]);
      }
    fclose(fp);
  }
```

运行结果：

FIR 数字滤波器的冲激响应（自适应滤波器的系数）为

0.0085903	−0.0003406	−0.0095127	−0.0003839
0.0118384	−0.0003677	−0.0131979	−0.0003455
0.0161255	−0.0003277	−0.0183730	−0.0003033
0.0225902	−0.0002857	−0.0267476	−0.0002495
0.0341659	−0.0002089	−0.0438677	−0.0000181
0.0632647	−0.0002407	−0.1047935	−0.0003181
0.3170237	0.4974658	0.3170271	−0.0002446
−0.1048183	−0.0003116	0.0627188	−0.0004307
−0.0439939	−0.0004199	0.0338166	−0.0004745
−0.0269246	−0.0004597	0.0222672	−0.0004954
−0.0185899	−0.0004819	0.0158042	−0.0005095
−0.0134538	−0.0004981	0.0114964	−0.0005236
−0.0098278	−0.0005175	0.0082612	

FIR 数字滤波器的单位冲激响应如图 3−4−6 所示，其幅频响应如图 3−4−7 所示。

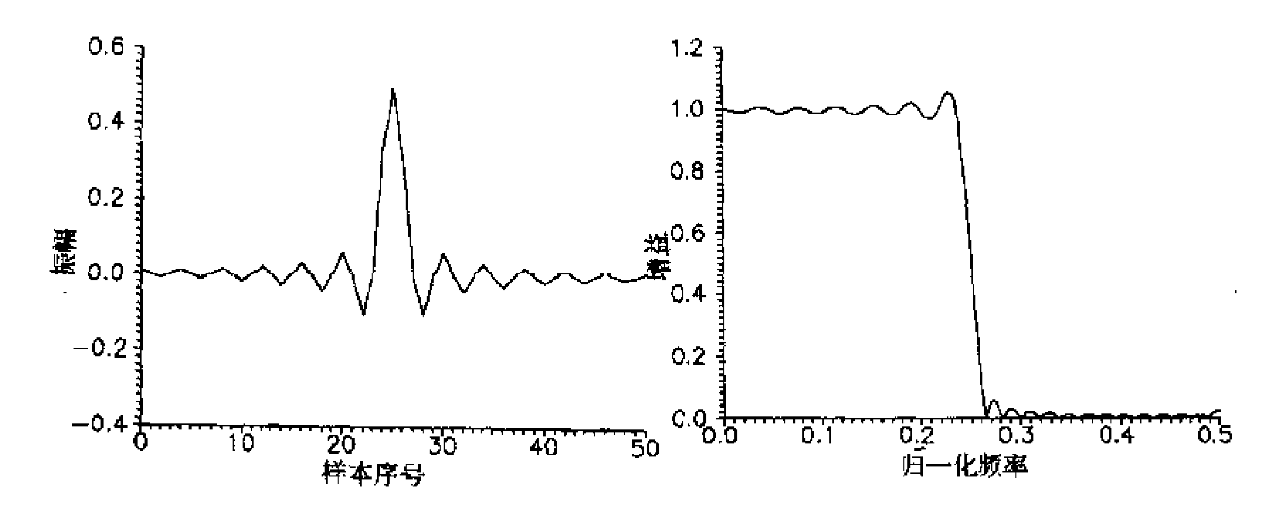


图 3—4—6　FIR 数字滤波器的单位冲激响应　　　图 3—4—7　FIR 数字滤波器的幅频响应

§4.5　递推最小二乘(RLS)自适应数字滤波

一、功　能

递推最小二乘(RLS)算法的自适应数字滤波器。

二、方法简介

设横向自适应滤波器的阶数为 M,滤波器的系数为 $w_i(n)(i=0,1,\cdots,M-1)$，输入为 $x(n)$，输出为 $y(n)$，理想输出为 $d(n)$，那么误差为

$$\begin{aligned} e(n) &= d(n) - y(n) \\ &= d(n) - W^T(n)X(n) \end{aligned}$$

其中输入向量为

$$X(n) = [x(n), x(n-1), \cdots, x(n-M+1)]^T$$

滤波器的加权系数向量为

$$W(n) = [w_0(n), w_1(n), \cdots, w_{M-1}(n)]^T$$

设遗忘因子为 λ,选择滤波器的系数 $W(n)$，使下式的性能函数为最小

$$\varepsilon(n) = \sum_{i=1}^{n} \lambda^{n-i} \, |e(i)|^2$$

从而得到正则方程为

$$\Phi(n)W(n) = \Theta(n)$$

其中 $M \times M$ 维相关矩阵为

$$\Phi(n) = \sum_{i=1}^{n} \lambda^{n-i} X(i) X^T(i)$$

$M \times 1$ 维互相关向量为

$$\Theta(n) = \sum_{i=1}^{n} \lambda^{n-i} X(i) d(i)$$

经过进一步推导，可得到递推最小二乘(RLS)算法如下：

1. 初始化

$$P(0) = \delta^{-1}I, \quad \delta \text{ 是很小的正常数}, I \text{ 是单位阵}$$
$$W(0) = 0$$

2. 对于 $n=1,2,\cdots,N$，进行如下计算

$$G(n) = \frac{\lambda^{-1}P(n-1)X(n)}{1+\lambda^{-1}X^T(n)P(n-1)X(n)}$$
$$\alpha(n) = d(n) - W^T(n-1)X(n)$$
$$W(n) = W(n-1) + G(n)\alpha(n)$$
$$P(n) = \lambda^{-1}P(n-1) - \lambda^{-1}G(n)X^T(n)P(n-1)$$

三、使用说明

1. 子函数语句

void rls(x,d,n,w,m,r)

2. 形参说明

x ——双精度实型一维数组，长度为 n。开始时存放输入信号，最后存放实际输出信号。

d ——双精度实型一维数组，长度为 n。理想输出信号。

n ——整型变量。输入信号的长度。

w ——双精度实型一维数组，长度为 m。自适应滤波器的加权系数。

m ——整型变量。自适应滤波器的长度(阶数−1)。

r ——双精度实型变量。遗忘因子，$0<r\leqslant1$。

四、子函数程序(文件名:rls.c)

```
#include "stdlib.h"
void rls(x,d,n,w,m,r)
int m,n;
double r,x[],d[],w[];
{ int i,j,k;
  double a,s,*g,*u,*px,*p;
  g = malloc(m*sizeof(double));
  u = malloc(m*sizeof(double));
  px = malloc(m*sizeof(double));
  p = malloc(m*m*sizeof(double));
  for (i=0;i<m;i++)
  for (j=0;j<m;j++)
    { p[i*m+j] = 0.0; }
  for (i=0;i<m;i++)
```

```
    { p[i * m+i] = 1.0e+8; }
  for (i=0;i<m;i++)
    { px[i] = 0.0; }
  for (k=0;k<n;k++)
    { px[0] = x[k];
      for (j=0;j<m;j++)
        { u[j] = 0.0;
          for (i=0;i<m;i++)
            { u[j] = u[j]+(1/r) * p[j * m+i] * px[i]; }
        }
      s = 1.0;
      for (i=0;i<m;i++)
        { s = s + u[i] * px[i]; }
      for (i=0;i<m;i++)
        { g[i] = u[i]/s;}
      x[k] = 0.0;
      for (i=0;i<m;i++)
        { x[k] = x[k] + w[i] * px[i]; }
      a = d[k] - x[k];
      for (i=0;i<m;i++)
        { w[i] = w[i] + g[i] * a; }
      for (j=0;j<m;j++)
      for (i=0;i<m;i++)
        { p[j * m+i] = (1/r) * p[j * m+i]-g[j] * u[i]; }
      for (i=(m-1);i>=1;i--)
        { px[i] = px[i-1]; }
    }
  free(g);
  free(u);
  free(px);
  free(p);
}
```

五、例 题

用递推最小二乘自适应算法进行 FIR 数字系统的辨识。FIR 数字系统为

$$y(i) = a_0x(i) + a_1x(i-1) + a_2x(i-2) + a_3x(i-3)$$

其中 $x(i)$ 为系统输入，本例中为零均值、单位方差的白噪声序列；$y(i)$ 为系统输出。选取参数为：输入与输出序列长度 $n=100$，系统系数 $a_0=2, a_1=-0.5, a_2=1.4, a_3=0.1$。

主函数程序(文件名:rls.m):

```
#include "stdio.h"
#include "gauss.c"
#include "rls.c"
main()
{ int i,m,n;
  long seed;
  double r,w[10],d[100],x[100];
  void rls();
  seed = 13579l;
  n = 100;
  for (i=0;i<n;i++)
    { x[i] = gauss(0.0,1.0,&seed); }
  d[0] = 2.0 * x[0];
  d[1] = 2.0 * x[1] - 0.5 * x[0];
  d[2] = 2.0 * x[2] - 0.5 * x[1] + 1.4 * x[0];
  for (i=3;i<n;i++)
    { d[i] = 2.0 * x[i] - 0.5 * x[i-1] + 1.4 * x[i-2] + 0.1 * x[i-3]; }
  m = 4;
  for (i=0;i<m;i++)
    { w[i] = 0.0; }
  r = 1.0;
  rls(x,d,n,w,m,r);
  printf("\n The Coefficients of FIR Digital System\n");
  for (i=0;i<m;i++)
    { printf("     w(%d) = %10.7lf\n",i,w[i]); }
}
```

运行结果:

FIR 数字系统的系数为

```
w(0) =      2.0000000
w(1) =     -0.5000000
w(2) =      1.4000000
w(3) =      0.1000000
```

第四篇　数字图像处理

第一章　图像基本运算

§1.1　图像读取、存储与显示

一、功　能

读取和存储 BMP 格式的图像文件，并在 VGA 显示器上进行显示。

二、方法简介

数字图像的每个象素通常用 8 个比特表示，因此，图像有 256 个灰度级，其范围为 0～255，其中 0 对应黑色，255 对应白色。

数字图像按一定的格式进行存储，BMP 格式就是最常用的格式之一。BMP 图像文件是 Microsoft Windows 系统的图像格式，它由 BMP 图像文件头和图像数据阵列两部分组成。

图像文件头包含了有关 BMP 图像的宽、高、压缩方法等信息，它的 C 语言结构如下：

```
struct BMP-Head
{ char                  ID[2] ;                  /* 总是 BM */
  unsigned  long        FileLength;              /* 文件大小,以字节为单位 */
  int                   Reserved1;               /* 必须为 0 */
  int                   Reserved2;               /* 必须为 0 */
  unsigned  long        ImagePosition;           /* 图像数据的起始位置 */
  unsigned  long        HeadLength;              /* 文件头的大小 */
  unsigned  long        ImageWidth;              /* 图像宽度,以象素为单位 */
  unsigned  long        ImageHeight;             /* 图像高度,以象素为单位 */
  int                   BitPlane ;               /* 必须为 1 */
  int                   ColorBit;                /* 每个象素的位数(1,4,8 或 24) */
  unsigned  long        CompressMethod;          /* 图像所用的压缩类型 */
  unsigned  long        ImageLength;             /* 图像字节数的多少 */
  unsigned  long        HorizontalDefinition;    /* 图像水平分辨率 */
  unsigned  long        VerticalDefinition;      /* 图像垂直分辨率 */
  unsigned  long        UsedColorNumber;         /* 图像实际使用的颜色数 */
  unsigned  long        ImportantColorNumber;    /* 重要的颜色数 */
```

} bmp-head;

图像数据阵列记录了图像的每个象素值。图像数据的存储是从图像的左下角开始逐行扫描图像,即从左到右,从下而上,将图像的象素值一一记录下来,从而形成了图像数据阵列。

在 VGA 显示器上用 256 色模式显示图像,图像的分辨率为 320×200。

可以将本篇的图像处理程序插入到本节的图像存取与显示程序中,这样就可以在 PC 微机上进行各种数字图像处理。

三、使用说明

1. 子函数语句

设置显示模式:

void SetScrMode(int);

获取显示模式:

int GetScrMode(void);

初始化图形系统:

void InitGraphics(void);

读取 BMP 图像文件头:

void ReadBMPHeadInfo(char *);

读取图像数据:

void load-data(char *, unsigned char huge *)

显示图像文件信息:

void DisplayImgHeadInfo(char *);

显示内存缓冲区中的图像:

void DisplayRoamImage(unsigned char huge *, long);

将内存缓冲区中的图像存入磁盘:

void SaveBufToFile(unsigned char huge *);

从磁盘中读取图像并放入内存缓冲区中,然后调用 DisplayRoamImage()进行显示:

void Display8BitBMPImage(char *);

在内存中开图像缓冲区:

unsigned char huge *alloc-mem(unsigned long)

注意:在下面给出的图像存取与显示程序 imagmain.c 中,已经插入了图像旋转子函数 rotate()。读者若插入其他图像处理子函数,可仿此进行。

2. 形参说明

四、子函数程序(文件名:imagmain.c)

```
#include "stdio.h"
#include "math.h"
#include "graphics.h"
```

```
#include "string.h"
#include "dos.h"
#include "stdlib.h"
#include "alloc.h"
#include "conio.h"
#include "bios.h"
#include "mem.h"
#include "dir.h"
#include "ctype.h"

#define   LEFT          75
#define   RIGHT         77
#define   UP            72
#define   DOWN          80
#define   ESC           0x011b
#define   ENTER         0x1c0d
#define   Home          71
#define   End           79
#define   Alt-x         45
#define   Ctrl-LEFT     115
#define   Ctrl-RIGHT    116
#define   Ctrl-Home     119
#define   Ctrl-End      117
#define   PgUp          73
#define   PgDn          81

typedef struct
{ int topx,topy;
  int bottomx,bottomy;
  char name[20];
  char color;
} BUTTON;

union KEY
{ int k;
  char c[2];
} key1;
```

```
struct BMP-Head
{ char                  ID[2] ;
  unsigned    long      FileLength;
  int                   Reserved1;
  int                   Reserved2;
  unsigned    long      ImagePosition;
  unsigned    long      HeadLength;
  unsigned    long      ImageWidth;
  unsigned    long      ImageHeight;
  int                   BitPlane ;
  int                   ColorBit;
  unsigned    long      CompressMethod;
  unsigned    long      ImageLength;
  unsigned    long      HorizontalDefinition;
  unsigned    long      VerticalDefinition;
  unsigned    long      UsedColorNumber;
  unsigned    long      ImportantColorNumber;
} bmp-head;

struct BMP-Palette
{ unsigned char palette[256][4];
} bmp-palette;

unsigned char huge  * i-img;
unsigned char huge  * o-img;
unsigned char huge  * shadow-buf;
unsigned int palette-size;
unsigned long img-row,img-col,line-size;
unsigned long w=256l,d=256l;
int INITmode;

void SetScrMode(int);
int GetScrMode(void);
void InitGraphics(void);
void ReadBMPHeadInfo(char * );
void DisplayImgHeadInfo(char * );
void DisplayRoamImage(unsigned char huge  * ,long);
void SaveBufToFile(unsigned char huge  * );
```

```
void Display8BitBMPImage(char * );

void InitGraphics(void)
{ int g-driver,g-mode,g-error;
  closegraph();
  g-driver = DETECT;
  g-error = 0;
  initgraph(&g-driver,&g-mode,"");
  g-error = graphresult();
  if (g-error<0)
    { outtextxy(30,120,"initgraph error!");
      exit(1);
      restorecrtmode();
    }
}

void ReadBMPHeadInfo(filename)
char * filename ;
{ FILE * fp;
  fp = fopen(filename,"rb");
  if ( fp == NULL )
    { printf("cannot open this file\n");
      exit(0);
    }
  fread(&bmp-head,sizeof(bmp-head),1,fp);
  if ((bmp-head. ID[0]!= 'B')&&(bmp-head. ID[1]!= 'M'))
    { printf("this file is not BMP file\n");
      fclose(fp);
      exit(0);
    }
  palette-size = bmp-head. ImagePosition - sizeof(bmp-head);
  fread(&bmp-palette,1,palette-size,fp);
  img-row = bmp-head. ImageHeight;
  img-col = bmp-head. ImageWidth;
  line-size = (bmp-head. ImageWidth * bmp-head. ColorBit + 31)/32 * 4;
  fclose(fp);
}
```

```
void load-data(filename,in-img)
char *filename;
unsigned char huge *in-img;
{ FILE *fp;
  int i,n;
  unsigned char huge *buffer;
  fp = fopen(filename,"rb");
  if (fp == NULL)
    { printf("Cannot open this file\n");
      exit(0);
    }
  fread(&bmp-head,sizeof(bmp-head),1,fp);
  if ((bmp-head.ID[0]!='B')&&(bmp-head.ID[1]!='M'))\=            { prinrf
("this file is not BMP file\n");
       fclose(fp);
       exit(0);
    }
  palette-size = bmp-head.ImagePosition - sizeof(bmp-head);
  fread(&bmp-palette,1,palette-size,fp);
  img-row = bmp-head.ImageHeight;
  img-col = bmp-head.ImageWidth;
  line-size = (bmp-head.ImageWidth * bmp-head.ColorBit + 31)/32 * 4;
  buffer = in-img;
  for (i=0;i<img-row;i++)
    { fread((void *)buffer,line-size,1,fp);
      buffer += line-size;
    }
  fclose(fp);
}

unsigned char huge *alloc-mem(buffer-size)
unsigned long buffer-size;
{ unsigned char huge *mem-ptr;
  if (buffer-size>farcoreleft())
    { printf("no enough memory");
      exit(0);
    }
  mem-ptr = (unsigned char huge *)farmalloc(buffer-size);
```

```
  if (mem-ptr == NULL)
     { printf("cannot allocate memory\n");
       exit(0);
     }
  return(mem-ptr);
}

void save-data(filename,buffer)
char *filename;
unsigned char huge *buffer;
{ int i,n;
  FILE *fp;
  fp = fopen(filename,"wb");
  if (fp == NULL) exit(0);
  fwrite(&bmp-head,sizeof(bmp-head),1,fp);
  n = bmp-head.ImagePosition - sizeof(bmp-head);
  fwrite(&bmp-palette,n,1,fp);
  for (i=0;i<img-row;i++)
     { fwrite((void *)buffer,line-size,1,fp);
       buffer += line-size;
     }
  fclose(fp);
}

void set-grey-palette(void)
{ int i,j;
  outportb(0x3c8,0x00);
  for (i=0;i<256;i++)
  for (j=2;j>=0;j--)
     { outportb(0x3c9,i>>2); }
}

void set-color-palette(palette,length)
unsigned char palette[][4];
int length;
{ int i,j,ch,k;
  outportb(0x3c8,0x00);
  for (i=0;i<length;i++)
```

```
  for (j=2;j>=0;j--)
    { ch = palette[i][j]>>2;
      outportb(0x3c9,ch);
    }
}

void SetScrMode(mode)
int mode;
{ union REGS reg;
  reg.h.ah = 0;
  reg.h.al = mode;
  int86(0x10,&reg,&reg);
}

int GetScrMode()
{ union REGS reg;
  reg.h.ah = 0xf;
  int86(0x10,&reg,&reg);
  return((int)reg.h.al);
}

void SaveBufToFile(result-image)
unsigned char huge *result-image;
{ int key1,key2;
  char save-filename[80];
  struct ffblk f;
  printf("Do you want to save this tranformed image ? (Y/N)\n");
  key1 = getch();
  if (toupper(key1) == 'Y')
    { do
        { key2=0;
          printf("\nPlease input the filename [.BMP]:");
          scanf("%s",save-filename);
          if (findfirst(save-filename,&f,0))
            { save-data(save-filename,result-image);
              printf("\n\n\aSave File Successfully !\n");
            }
          else
```

```
            { printf("\nImage %s has existed ! \n",save-filename);
              printf("OverWrite/Abort/Retry-->\a\n");
              key2 = toupper(getche());
              if (key2== 'O')
                { save-data(save-filename,result-image);
                  printf("\n\n\aOverWrite Successfully !\n");
                }
            }
        } while (key2== 'R');
    }
  printf("\nPress any key to continue ...");
  getch();
}

void DisplayImgHeadInfo(filename)
char *filename;
{ unsigned long buffer-size;
  ReadBMPHeadInfo(filename);
  buffer-size = img-row * line-size;
  printf("\n\n%s Image Information:\n\n",filename);
  printf("Height : %ld\n",img-row);
  printf("Width : %ld\n",img-col);
  printf("Line Size : %ld\n",line-size);
  printf("Image Length : %ld\n",buffer-size);
  printf("Palette size : %d\n\n",palette-size);
  printf("Press any key to continue ...\n");
  getch();
}

void DisplayRoamImage(image-data,img-row)
unsigned char huge *image-data;
long img-row;
{ int i,j,k,seg,off,row-remain,col-remain;
  int disp-row,disp-col,disp-row1;
  unsigned char huge *data;
  if ((row-remain=(int)img-row-200)>0)
    { disp-row = 200;}
  else
```

```
  { disp-row = img-row;
    row-remain = 0;
  }
if ((col-remain = (int)img-col-320)>0)
  { disp-col = 320; }
else
  { disp-col = img-col;
    col-remain = 0;
  }
j = 0;
k = 0;
do
  { data = image-data + (img-row-j) * line-size + k;
    if (j < 0)
      { data = image-data + img-row * line-size + k;
        clrscr();
        for (i=(-j);i<disp-row;i+-)
          { data -= line-size;
            seg = FP-SEG(data);
            off = FP-OFF(data);
            movedata(seg,off,0xa000,(long)i * 320,disp-col);
          }
      }
    else
      { if (j < row-remain)
          { for (i=0;i<disp-row;i++)
              { data  -= line-size;
                seg = FP-SEG(data);
                off = FP-OFF(data);
                movedata(seg,off,0xa000,(long)i * 320,disp-col);
              }
          }
        else
          { disp-row1 = disp-row - (j - row-remain);
            clrscr();
            for (i=0;i<disp-row1;i++)
              { data -= line-size;
                seg = FP-SEG(data);
```

```
                off = FP-OFF(data);
                movedata(seg,off,0xa000,(long)i * 320,disp-col);
              }
          }
      }
    key1.k = bioskey(0);
    if (key1.c[0] == 0)
      { switch(key1.c[1])
          { case DOWN :
              j++;
              break;
            case UP :
              j--;
              break;
            case End :
              if (j == row-remain)
                { printf("\a");
                  break;
                }
              else
                { j = row-remain;
                  break;
                }
              case Home :
                if (j == 0)
                  { printf("\a");
                    break;
                  }
                else
                  { j = 0;
                    break;
                  }
              case RIGHT:
                if (k == col-remain)
                  { printf("\a");
                    break;
                  }
                else
```

```
                { k++;
                  break;
                }
            case LEFT :
              if ( k == 0 )
                { printf("\a");
                  break;
                }
              else
                { k--;
                  break;
                }
            case Ctrl-Home:
              if ( k == 0 )
                { printf("\a");
                  break;
                }
              else
                { k = 0;
                  break;
                }
            case Ctrl-End :
              if (k == col-remain)
                { printf("\a");
                  break;
                }
              else
                { k = col-remain;
                  break;
                }
            case Alt-x:
              return;
        }
    }
  } while ( 1 );
}

void Display8BitBMPImage(filename)
```

```
char *filename;
{ ReadBMPHeadInfo(filename);
  i-img = alloc-mem(img-row * line-size);
  load-data(filename,i-img);
  DisplayImgHeadInfo(filename);
  SetScrMode(0x13);
  set-color-palette(bmp-palette. palette,palette-size/4);
  DisplayRoamImage(i-img,img-row);
  SetScrMode(INITmode);
}

/* ------------------------ */
/* The following is image processing subfunction */
/* ------------------------ */
#include "math. h"
#define ff(i,j) ip[(long)(i) * lx+j]
#define gg(i,j) jp[(long)(i) * lx+j]
void rotate (ip,jp,lx,ly,theta)
unsigned char huge *ip, *jp;
unsigned long lx,ly;
double theta;
{ int temp;
   unsigned long i,j;
   long x,y,oldx,oldy;
   double d0,d1,fx,fy,pi2,temp1,temp2;
   double s,c,angle;
   pi2 = 8 * atan(1. 0);
   angle = (theta/360. 0) * pi2;
   for (i=0l;i<ly;i++)
   for (j=0l;j<lx;j++)
      { gg(i,j) = 0; }
   for (y=0l;y<ly;y++)
   for (x=0l;x<lx;x++)
      { c = cos(angle);
        s = sin(angle);
        fx =((double)x - (double)(lx-1l)/2. 0 ) * c -
             ((double)y - (double)(ly-1l)/2. 0 ) * s + (lx-1l)/2. 0;
        fy = ((double)x - (double)(lx-1l)/2. 0 ) * s +
```

```
            ((double)y - (double)(ly-1l)/2.0) * c + (ly-1l)/2.0;
        oldx = (long)floor(fx);
        oldy = (long)floor(fy);
        d0 = fy - oldy;
        d1 = fx - oldx;
        if ((oldy < ly-1l) && (oldy >= 0l))
        if ((oldx < lx-1l) && (oldx >= 0l))
          { temp1 = (1-d1) * ff(oldy,oldx) + d1 * ff(oldy,oldx+1l);
            temp2 = (1-d1) * ff(oldy+1l,oldx) + d1 * ff(oldy+1l,oldx+1l);
            temp = (1-d0) * temp1 + d0 * temp2;
            gg(y,x) = temp;
          }
      }
}
/* ---------------------- */
```

五、例　题

下面给出图像读取、显示与存储的主函数。图像处理子函数的调用语句可插入该程序中。例如，该主函数已插入了图像旋转子函数的调用语句。执行该程序，即可读取、显示和存储BMP格式的图像。

主函数程序(文件名:imagmain.m):

```
#include "imagmain.c"
main()
{ char filename[80];
  int key;
  INITmode=GetScrMode();
  closegraph();
  printf("Please input BMP-image filename[*.bmp]:");
  scanf("%s",filename);
  Display8BitBMPImage(filename);
  clrscr();
  o-img=alloc-mem(img-row * line-size);
  printf("Waiting...\n\a");
/* ------------------- */
/* 此处调用图像处理子函数        */
  rotate (i-img,o-img,256l,256l,90);
/* ------------------- */
  printf("\a");
```

```
  SetScrMode(0x13);
  set-color-palette(bmp-palette. palette,palette-size/4);
  DisplayRoamImage(o-img,img-row);
  clrscr();
  SetScrMode(INITmode);
  SaveBufToFile(o-img);
  clrscr();
  SetScrMode(0x13);
  set-color-palette(bmp-palette. palette,palette-size/4);
  clrscr();
  SetScrMode(INITmode);
  farfree((void far  * )o-img);
  farfree((void far  * )i-img);
  SetScrMode(INITmode);
}
```

运行结果：

输入图像：测试女郎“Lena”图像(文件名:Lena. bmp)，显示结果见图 4—1—1。

§1.2 图像旋转

一、功 能

将图像进行任意角度的旋转。

二、方法简介

设原图像为 $f(x,y)$，旋转后的图像为 $g(x',y')$，旋转角度为 α，旋转中心坐标为 (x_0,y_0)，那么旋转变换方程为

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x'-x_0 \\ y'-y_0 \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

对于 $M\times N$ 图像，旋转中心通常选为图像的中心，于是有

$$x_0=\frac{M}{2},\quad y_0=\frac{N}{2}$$

由于经旋转变换得到的原图像坐标(x,y)一般不是整数值，因此，旋转后图像中(x',y')处的灰度值，应由原图像(x,y)周围的像素的灰度值来确定。以处采用的是四点线性插值法。

三、使用说明

1. 子函数语句

```
void rotate(ip,jp,lx,ly,theta)
```

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

theta ——双精度实型变量。旋转角度，单位为度。

四、子函数程序(文件名:rotate.c)

```
#include "math.h"
#define ff(i,j) ip[(long)(i) * lx+j]
#define gg(i,j) jp[(long)(i) * lx+j]
void rotate (ip,jp,lx,ly,theta)
unsigned char huge  * ip, * jp;
unsigned long lx,ly;
double theta;
{ int temp;
  unsigned long i,j;
  long x,y,oldx,oldy;
  double d0,d1,fx,fy,pi2,temp1,temp2;
  double s,c,angle;
  pi2 = 8 * atan(1.0);
  angle = (theta/360.0) * pi2;
  for (i=0l;i<ly;i++)
  for (j=0l;j<lx;j++)
    { gg(i,j) = 0; }
  for (y=0l;y<ly;y++)
  for (x=0l;x<lx;x++)
    { c = cos(angle);
      s = sin(angle);
      fx =((double)x - (double)(lx-1l)/2.0 ) * c -
           ((double)y - (double)(ly-1l)/2.0) * s + (lx-1l)/2.0;
      fy =((double)x - (double)(lx-1l)/2.0) * s +
           ((double)y - (double)(ly-1l)/2.0) * c + (ly-1l)/2.0;
      oldx = (long)floor(fx);
      oldy = (long)floor(fy);
```

```
        d0 = fy - oldy;
        d1 = fx - oldx;
        if ((oldy < ly-1l) && (oldy >= 0l))
        if ((oldx < lx-1l) && (oldx >= 0l))
          { temp1 = (1-d1) * ff(oldy,oldx) + d1 * ff(oldy,oldx+1l);
            temp2 = (1-d1) * ff(oldy+1l,oldx) + d1 * ff(oldy+1l,oldx+1l);
            temp = (1-d0) * temp1 + d0 * temp2;
            gg(y,x) = temp;
          }
      }
}
```

五、例　题

将图像旋转 90 度，调用语句：

```
rotate(i-img,o-img,256l,256l,90);
```

输入图像："Lena"图像(文件名:lena.bmp)，图像大小为 256×256，见图 4—1—1。

输出图像：见图 4—1—2。



图 4—1—1　测试女郎"Lena"图像



图 4—1—2　旋转 90°的"Lena"图像

§1.3 图像灰度级直方图的计算

一、功　能

计算图像的灰度级直方图。

二、方法简介

将图像中所有像素按其灰度值的大小进行计数，从而得到像素数量随灰度级变化的函数，这就是图像的灰度级直方图。

设图像为 $f(x,y)$，灰度级直方图为 hist(i)，那么直方图的计算方法如下：

1. 初始化：

hist$(i) = 0$ ， $i=0,1,\cdots,255$

2. 对于所有 x 和 y，进行如下计算：

$k = f(x,y)$

hist(k) = hist$(k) + 1$

三、使用说明

1. 子函数语句

void histgram(ip,lx,ly,hist)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

hist——无符号长整型一维数组，长度为 256。图像的灰度级直方图。

四、子函数程序(文件名:histgram.c)

```
#define ff(i,j) ip[(i)*lx+(j)]
void histgram(ip,lx,ly,hist)
unsigned char huge *ip;
unsigned long lx,ly,hist[256];
{ int i,j,k;
  unsigned long max;
  double t;
  static char *textstring[6]={"0","50","100","150","200","250"};
  static char *topic="HISTOGRAM";
  for (i=0;i<256;i++)
```

```
    { hist[i] = 0; }
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { k = ff(i,j);
      hist[k]++;
    }
  max = 0;
  for (i=0;i<256;i++)
    { max = (max < hist[i]) ? hist[i] : max; }
  InitGraphics();
  setcolor(GREEN);
  moveto(20,300);
  lineto(600,300);
  lineto(595,305);
  line(600,300,595,295);
  line(20,300,20,20);
  line(20,20,15,25);
  line(20,20,25,25);
  for (i=0;i<6;i++)
    { outtextxy(21+i * 100,310,textstring[i]); }
  setcolor(LIGHTRED);
  for (i=0;i<256;i++)
    { t = (double)hist[i]/max;
      k = t * 250;
      moveto(21+i * 2,300);
      lineto(21+i * 2,300-k);
    }
  settextstyle(DEFAULT-FONT,HORIZ-DIR,4);
  setcolor(CYAN);
  outtextxy(190,400,topic);
}
```

五、例　题

计算“Lena”图像的灰度级直方图，调用语句：

　histgram(i-img,256l,256l,hist);

输入图像：“Lena”图像(文件名：lena.bmp)，图像大小为 256×256，见图 4—1—1。

输出图像：见图 4—1—3。

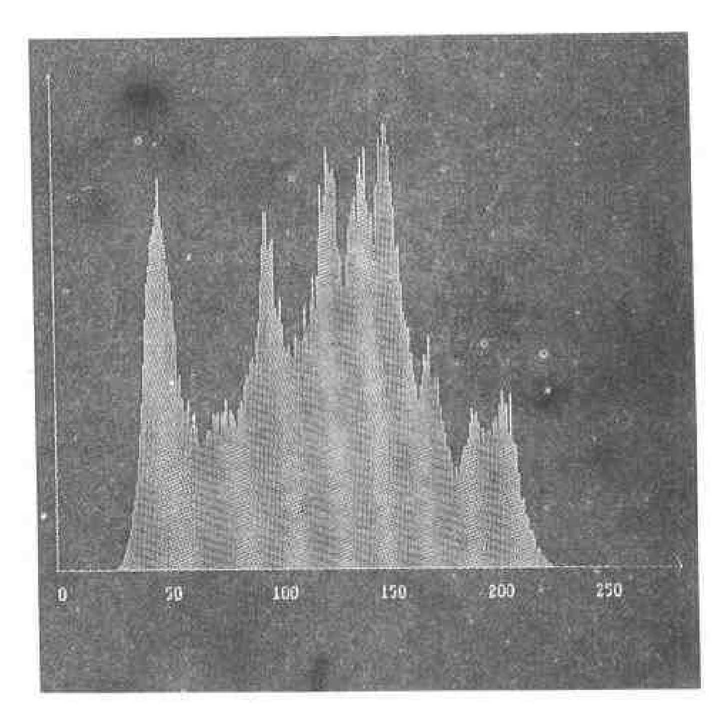


图 4—1—3 “Lena”图像的直方图

§1.4 图像二值化的固定阀值法

一、功 能

用固定阀值法对图像进行黑白二值化处理。

二、方法简介

图像二值化是通过设定某个阀值，把具有灰度级的图像变换成只有两个灰度级的黑白图像。设输入图像为 $f(x,y)$，二值化后的图像为 $g(x,y)$，阀值为 T，那么图像二值化的方法为

$$g(x,y)=\begin{cases}255 & , \quad f(x,y)\geqslant T\\ 0 & , \quad f(x,y)<T\end{cases}$$

三、使用说明

1. 子函数语句

void thresh(ip,jp,lx,ly,threshold)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)

点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

threshold——无符号字符型变量。图像二值化的阈值。

四、子函数程序(文件名:thresh.c)

```
#define ff(i,j) ip[(i) * lx+(j)]
#define gg(i,j) jp[(i) * lx+(j)]
void thresh(ip,jp,lx,ly,threshold)
unsigned char huge  *ip, *jp;
unsigned long lx,ly;
unsigned char threshold;
{ int i,j;
  for (i=0l;i<ly;i++)
  for (j=0l;j<lx;j++)
    { if (ff(i,j) >= threshold)
        { gg(i,j) = 255; }
      else
        { gg(i,j) = 0; }
    }
}
```

五、例　题

将"Lena"图像二值化,阈值选为 110,调用语句:

thresh(i-img,o-img,256l,256l,110);

输入图像:"Lena"图像(文件名:lena.bmp),图像大小为 256×256,见图 4—1—1。

输出图像:见图 4—1—4。



图 4—1—4　固定阈值二值化后的图像

§1.5　图像二值化的自适应阈值法

一、功　能

用自适应阈值法对图像进行黑白二值化处理。

二、方法简介

用 Otsu 提出的类判别分析法自动选择阈值，然后用该阈值进行图像二值化。

设输入图像为 $f(x,y)$，二值化后的图像为 $g(x,y)$，阈值为 T，那么图像二值化的自适应阈值法如下：

1. 用本章§1.3节的方法，计算输入图像灰度级的归一化直方图，用 $h(i)$ 表示。
2. 计算灰度均值 μ_T

$$\mu_T = \sum_{i=0}^{255} ih(i)$$

3. 计算直方图的零阶累积矩 $\omega(k)$ 和一阶累积矩 $\mu(k)$

$$\omega(k) = \sum_{i=0}^{k} h(i) \qquad k = 0,1,\cdots,255$$
$$\mu(k) = \sum_{i=0}^{k} ih(i)$$

4. 计算类分离指标

$$\sigma_B(k) = \frac{[\mu_T\,\omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]} \qquad k = 0,1,\cdots,255$$

5. 求 $\sigma_B(k)(k=0,1,\cdots,255)$ 的最大值，并将其所对应的 k 值作为最佳阈值 T。
6. 对输入图像进行二值化处理

$$g(x,y) = \begin{cases} 255, & f(x,y) \geqslant T \\ 0, & f(x,y) < T \end{cases}$$

三、使用说明

1. 子函数语句

void adpthres(ip,jp,lx,ly)

2. 形参说明

ip ——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:adpthres.c)

```
#define ff(i,j) ip[(i)*lx+(j)]
#define gg(i,j) jp[(i)*lx+(j)]
void adpthres(ip,jp,lx,ly)
unsigned char huge *ip,*jp;
```

```
unsigned long lx,ly;
{ int i,j,k,threshold;
  double uk,b,wk,q,s,ut,max,eps,hist[256];
  eps = 1.0e-10;
  for (i=0;i<256;i++)
    { hist[i] = 0.0; }
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { k = ff(i,j);
      hist[k]++;
    }
  s = 0.0;
  for (i=0;i<256;i++)
    { s += hist[i]; }
  for (i=0;i<256;i++)
    { hist[i] = hist[i]/s; }
  ut = 0.0;
  for (i=0;i<256;i++)
    { ut = ut+i * hist[i]; }
  uk = 0.0;
  wk = 0.0;
  max = 0.0;
  for (i=0;i<256;i++)
    { wk = wk + hist[i];
      uk = uk + i * hist[i];
      b = ut * wk - uk;
      q = wk * (1.0-wk);
      if (q <= eps) continue;
      b = b * b/q;
      if ( b > max )
        { max = b;
          threshold = i;
        }
    }
  printf("\n\n\n        Threshold = %d\n",threshold);
  printf("\n        Press any key to continue ...\n");
  getch();
  for (i=0l;i<ly;i++)
```

```
for (j=0l;j<lx;j++)
  { if (ff(i,j) >= threshold)
      { gg(i,j) = 255; }
    else
      { gg(i,j) = 0; }
  }
}
```

五、例　题

将"Lena"图像二值化，调用语句：

adpthres(i-img,o-img,256l,256l);

输入图像："Lena"图像(文件名：lena.bmp)，图像大小为 256×256，见图 4—1—1。

输出图像：见图 4—1—5。



图 4—1—5　自适应阈值二值化后的图像

第二章　图像增强

§2.1　图像直方图均衡

一、功　能

通过直方图均衡化进行图像增强。

二、方法简介

用灰度变换函数对输入图像直方图进行修正,使修正后的图像直方图趋向均匀分布，从而使图像灰度级的动态范围增大，达到改善图像质量的目的，这就是图像的直方图均衡。

设 r_k 为原始图像的第 k 级灰度，s_k 为变换后图像的第 k 级灰度，那么图像直方图均衡的具体方法如下：

1. 用本篇§1.3节的方法,计算原始图像的灰度级直方图,用 $n(i)(i=0,1,\cdots,255)$ 表示。

2. 计算灰度级变换函数 $T(r_k)$

$$s_k = T(r_k) = \sum_{i=0}^{k}\frac{n(i)}{N} \qquad k = 0,1,\cdots,255$$

其中 N 是图像的像素总数。

3. 根据灰度级变换函数 $T(r_k)$ 完成图像的灰度级变换。

由于 $\frac{n(i)}{N}$ 只是图像灰度级概率密度函数的近似，因此经过直方图均衡后，只能得到接近平坦的直方图。

三、使用说明

1. 子函数语句

```
void histeq(ip,jp,lx,ly)
```

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:histeq.c)

```
#define ff(i,j) ip[(long)(i) * lx+j]
#define gg(i,j) jp[(long)(i) * lx+j]
void histeq(ip,jp,lx,ly)
unsigned char huge *ip, *jp;
unsigned long lx,ly;
{ int i,j,k,medvar,medvar2,ttv;
  long maxvalue,maxvalue2;
  unsigned long level[256], level1[256], level2[256];
  for (i=0;i<256;i++)
    { level[i] = level1[i] = level2[i]=0; }
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { ttv = (int)ff(i,j);
      level[ttv]++;
    }
  level1[0] = level[0] * 255;
  for (i=0;i<256;i++)
    { level1[i] = level1[i-1] + level[i] * 255; }
  for (i=0;i<256;i++)
    { level1[i] = ((medvar2=level1[i]/(lx * ly))>255) ? 255 : medvar2; }
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { medvar = level1[ff(i,j)];
      gg(i,j) = medvar;
      level2[medvar]++;
    }
}
```

五、例 题

将蔬菜“Veget”图像进行直方图均衡，调用语句：

histeq(i-img,o-img,256l,256l);

输入图像：蔬菜“Veget”图像(文件名:veget.bmp)，图像大小为 256×256，见图 4—2—1。

输出图像：见图 4—2—2。



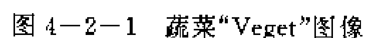


图 4—2—1　蔬菜“Veget”图像



图 4—2—2　直方图均衡后的蔬菜图像

§2.2　中值滤波

一、功　能

用快速中值滤波方法对图像进行滤波，去除图像中的噪声。

二、方法简介

中值滤波是一种能有效地抑制图像中噪声的非线性信号处理技术。它有两个主要优点：(1) 可以有效地抑制尖锐和长拖尾噪声；(2) 在滤波时能够保护信号中尖锐的跳变或边界。

设 x_{ij}是输入图像在坐标(i,j)处的像素，在其周围开一个 $m\times n$ 的矩形窗口。假设窗口从左向右水平扫描，而后再回到下一行重复扫描。对每个窗口内的所有像素按其灰度值的大小进行排序，求出中值 x_M，然后用中值 x_M 替换 x_{ij}，这就是图像的中值滤波。

在二维中值滤波时，利用窗口的滑动性质，可以推导出快速中值滤波算法。具体步骤如下：

1. 计算第一个窗口内图像元素的灰度级直方图 hist$[i]$$(i=0,1,\cdots,255)$，并找出中值 mdn，然后计算出灰度值小于中值 mdn 的像素数 ltmdn。以后，窗口每向右扫描一步，都对这三个量进行更新。

2. 将窗口向右移动一个单位时，原先窗口内最左边一列的每个像素的灰度级 g 被删去。因此，直方图 hist 和 ltmdn 更新为

$$\text{hist}[g] \leftarrow \text{hist}[g] - 1$$

和

$$\text{ltmdn} \leftarrow \text{ltmdn} - 1 \quad ，当\ g<\text{mdn}$$

同样，现在窗口内最右边一列的每个像素的灰度级 g 被加进。这时直方图 hist 和 ltmdn

按下述方法更新

$$\text{hist}[g] \leftarrow \text{hist}[g] + 1$$

和

$$\text{ltmdn} \leftarrow \text{ltmdn} + 1 \quad \text{，当 } g < \text{mdn}$$

这样完成以后，hist 就是当前窗口的直方图，而 ltmdn 是当前窗口内灰度值小于原先窗口中值的像素数。

3. 计算当前窗口的中值

设阀值 $T=(m\times n-1)/2$。我们，比较 ltmdn 和 T，然后分为两种情况进行处理：

情况Ⅰ：ltmdn>T，这表示 mdn 大于当前窗口的中值。此时进行如下计算：

$$\text{mdn} \leftarrow \text{mdn} - 1$$

$$\text{ltmdn} \leftarrow \text{ltmdn} - \text{hist}[\text{mdn}]$$

直到

$$\text{ltmdn} \leqslant T$$

情况Ⅱ：ltmdn≤T，这表示 mdn 小于或等于当前窗口的中值。这时检验

$$\text{ltmdn} + \text{hist}[\text{mdn}] \leqslant T$$

如果不成立，则 mdn 正好是当前窗口的中值。如果成立，它表示 mdn+1 仍然小于或等于所要求的中值，这时计算

$$\text{ltmdn} \leftarrow \text{ltmdn} + \text{hist}[\text{mdn}]$$

$$\text{mdn} \leftarrow \text{mdn} + 1$$

然后再进行检验。如此反复计算，直到 ltmdn+hist[mdn] > T 为止。

三、使用说明

1. 子函数语句

void medfilt(ip,jp,lx,ly,w)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

w——整型变量。方形窗口宽度。

四、子函数程序(文件名：medfilt.c)

```
#define gg(i,j) jp[(long)(i) * lx+(long)(j)]
void medfilt(ip,jp,lx,ly,w)
unsigned char huge *ip, *jp;
unsigned long lx,ly;
```

```
int w;
{ unsigned int hist[256], win[100];
  int i,j,k,l,ii,jj,m,n,mv,q;
  if (( w < 3)||(w%2 !=1))
    { printf("\nILLEGAL INPUT !\n");
      exit(0);
    }
  k = (w-1)/2;
  for (i=k;i<lx-k;i++)
  for (j=k;j<lx-k;j++)
    { if (j == k)
        { m = 0;
          for (jj=j-k;jj<=j+k;jj++)
          for (ii=i-k;ii<=i+k;ii++)
            { win[m] = ip[ii * ly+jj];
              m++;
            }
          for (ii=0;ii<w * w-1;ii++)
          for (jj=0;jj<w * w-1-ii;jj++)
            { if (win[jj] < win[jj+1])
                { m = win[jj];
                  win[jj] = win[jj+1];
                  win[jj+1] = m;
                }
            }
          mv = win[(w * w+1)/2];
          l = 0;
          for (q=0;q<256;q++)
            { hist[q] = 0; }
          for (ii=0;ii<w * w;ii++)
            { m = win[ii];
              hist[m] = hist[m] + 1;
              if (m < mv) l++;
            }
        }
      else
        { for (ii=i-k;ii<=i+k;ii++)
            { m = ip[ii * ly+j+k];
```

```
                n = ip[ii * ly+j-k-1];
                hist[m] = hist[m] + 1;
                hist[n]=hist[n]-1;
                if (n < mv) l--;
                if (m < mv) l++;
              }
          }
        while ( l >= (w * w+1)/2 )
          { l = l - hist[mv-1];
            mv = mv - 1;
          }
        while ( l+hist[mv] < (w * w+1)/2 )
          { l = l + hist[mv];
            mv = mv + 1;
          }
        gg(i,j) = mv;
      }
  }
```

五、例　题

对受噪声污染的“Lena”图像进行中值滤波，调用语句：

medfilt(i-img,o-img,256l,256l);

输入图像：受噪声污染的“Lena”图像(文件名:lena10.bmp)，图像大小为256×256，见图4—2—3。

输出图像：见图4—2—4。



图4—2—3　含有噪声的图像



图4—2—4　中值滤波后的图像

§2.3 图像锐化

一、功　能

用空间高通滤波方法进行图像锐化。

二、方法简介

图像锐化的目的是加强图像的轮廓和边缘，常用的方法之一是进行空间高通滤波。此处选用的高通滤波器的单位冲激响应阵列为

$$H = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

即

$$\begin{aligned} g(x,y) &= f(x,y) - \nabla^2 f(x,y) \\ &= 5f(x,y) - f(x-1,y) - f(x-1,y) - f(x,y+1) - f(x,y-1) \end{aligned}$$

其中 $f(x,y)$ 是输入图像，$g(x,y)$ 是输出图像，$\nabla^2 f(x,y)$ 是输入图像二次微分的数字拉普拉斯算子。上式实质上也是用拉普拉斯算子进行图像锐化。

三、使用说明

1. 子函数语句

void sharpen(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:sharpen.c)

```
#define ff(i,j) ip[(long)(i)*lx+(j)]
#define gg(i,j) jp[(long)(i)*lx+(j)]
void sharpen(ip,jp,lx,ly)
unsigned char huge *ip, *jp;
unsigned long lx,ly;
{ unsigned long i,j;
  int temp;
```

```
for (i=1l;i<ly-1l;i++)
for (j=1l;j<lx-1l;j++)
  { temp = 5 * (int)ff(i,j) - (int)ff(i-1,j) - (int)ff(i+1,j)
          - (int)ff(i,j+1) - (int)ff(i,j-1);
    if (temp > 255)
      { temp = 255; }
    else if(temp < 0 )
      { temp = 0; }
    gg(i,j) = temp;
  }
}
```

五、例　题

对"Lena"图像进行锐化处理，调用语句：

sharpen(i-img,o-img,256l,256l);

输入图像："Lena"图像（文件名：lena.bmp），图像大小为 256×256，见图 4－1－1。

输出图像：见图 4－2－5。



图 4－2－5　锐化处理后的图像

§2.4　图像平滑

一、功　能

用邻域平均方法进行图像平滑。

二、方法简介

设输入图像为 $f(x,y)$，输出图像为 $g(x,y)$。若点 (x,y) 的邻域为 S，那么 4 邻域的坐标集合为

$$S = \{(x,y+1),(x,y-1),(x+1,y),(x-1,y)\}$$

8 邻域的坐标集合为

$$S = \{(x-1,y-1),(x,y-1),(x+1,y-1),(x-1,y),\\(x+1,y),(x-1,y+1),(x,y+1),(x+1,y+1)\}$$

图像平滑的邻域平均方法为

$$g(x,y)=\begin{cases}\frac{1}{M}\sum_{(x,y)\in S}f(x,y) & ,\quad 当\left|f(x,y)-\frac{1}{M}\sum_{(x,y)\in S}f(x,y)\right|\geqslant T\\ f(x,y) & ,\quad 其他\end{cases}$$

其中 T 是阈值。

三、使用说明

1. 子函数语句

void smooth(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:smooth.c)

```
#define ff(i,j) ip[(long)(i) * lx+(j) ]
#define gg(i,j) jp[(long)(i) * lx+(j) ]
void smooth(ip,jp,lx,ly)
unsigned char huge  * ip, * jp;
unsigned long lx,ly;
{ int thresh,temp,choice;
  unsigned long i,j;
  puts("\nPlease Choose: 1-4 adjacent points;   2-8 adjacent points ");
  scanf(" %d",&choice);
  puts("\nPlease input smooth threshold ");
  scanf(" %d",&thresh);
  switch (choice)
    { case 1:
        for (i=1l;i<ly-1l;i++)
        for (j=1l;j<lx-1l ;j++)
          { temp = (int)ff(i+1,j) + (int)ff(i-1,j) +
                   (int)ff(i,j+1) + (int)ff(i,j-1);
            temp/=4;
            if (abs(ff(i,j)-temp)>thresh)
              { gg(i,j) = temp; }
```

```
            else
               { gg(i,j) = ff(i,j); }
         }
      case 2:
        for (i=1l;i<ly-1l;i++)
        for (j=1l;j<lx-1l;j++)
          { temp = (int)ff(i+1,j) + (int)ff(i-1,j) +
                   (int)ff(i,j+1) + (int)ff(i,j-1) +
                   (int)ff(i-1,j-1) + (int)ff(i-1,j+1) +
                   (int)ff(i+1,j-1) + (int)ff(i+1,j+1);
            temp/= 8;
            if (abs(ff(i,j)-temp)>thresh)
               { gg(i,j) = temp; }
            else
               { gg(i,j) = ff(i,j); }
          }
    }
}
```

五、例　题

对"Lena"图像进行平滑，调用语句：

smooth(i-img,o-img,256l,256l);

输入图像："Lena"图像(文件名:lena.bmp)，图像大小为 256×256，见图 4－1－1。

输出图像：见图 4－2－6。



图 4－2－6　平滑后的图像

第三章　图像边缘检测

§3.1　Roberts 算子边缘检测

一、功　能

用 Roberts 算子进行图像的边缘检测。

二、方法简介

设输入图像为 $f(x,y)$，输出图像为 $g(x,y)$，则第一种 Roberts 微分运算定义为

$$g(x,y) = |f(x+1,y) - f(x,y)| + |f(x,y+1) - f(x,y)|$$

第二种 Roberts 微分运算定义为

$$g(x,y) = |f(x+1,y+1) - f(x,y)| + |f(x+1,y) - f(x,y+1)|$$

三、使用说明

1. 子函数语句

void roberts(ip,jp,lx,ly,type)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

type——整型变量。type＝1，表示第一种 Roberts 算子；type＝2，表示第二种 Roberts 算子。

四、子函数程序(文件名:roberts.c)

```
#include "math.h"
#define ff(x,y) ip[lx * (y)-(x)]
#define gg(x,y) jp[lx * (y)+(x)]
void roberts(ip,jp,lx,ly,type)
unsigned char huge  *ip, *jp;
unsigned long lx,ly;
```

```
int type;
{ int ix,iy,vx,vy,v2;
  for (ix=0;ix<lx;ix++)
    { gg(ix,0) = 0;
      gg(ix,ly-1) = 0;
    }
  for (iy=1;iy<ly;iy++)
    { gg(0,iy) = 0;
      gg(lx-1,iy) = 0;
    }
  if ( type == 1)
    { for (ix=1;ix<lx-1;ix++)
        for (iy=1;iy<ly-1;iy++)
          { vx = ff(ix+1,iy) - ff(ix,iy);
            vy = ff(ix,iy+1) - ff(ix,iy);
            gg(ix,iy) = abs(vx) + abs(vy);
          }
    }
  else
    { for (ix=1;ix<lx-1;ix++)
        for (iy=1;iy<ly-1;iy++)
          { vx = ff(ix,iy) - ff(ix+1,iy+1);
            vy = ff(ix+1,iy) - ff(ix,iy+1);
            gg(ix,iy) = abs(vx) + abs(vy);
          }
    }
}
```

五、例　题

用第二种 Roberts 微分运算对猴子“Monkey”图像进行边缘检测，调用语句：

```
roberts(i-img,o-img,256l,256l,2);
```

输入图像：猴子“Monkey”图像（文件名：monkey.bmp），图像大小为 256×256，见图 4－3－1。

输出图像：见图 4－3－2。

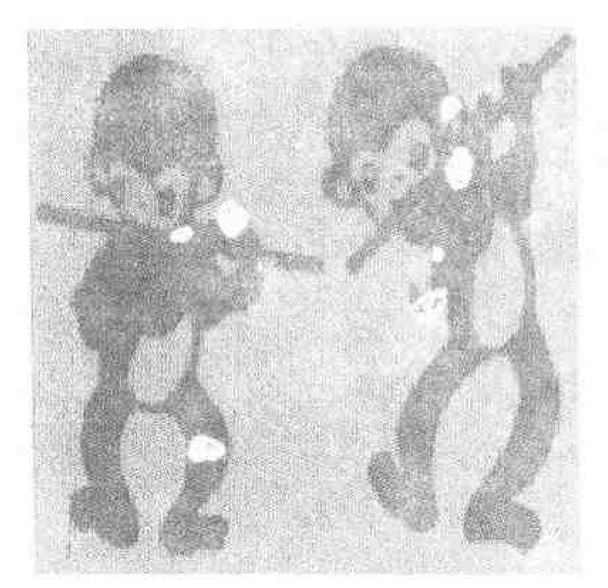


图 4—3—1 猴子"Monkey"图像

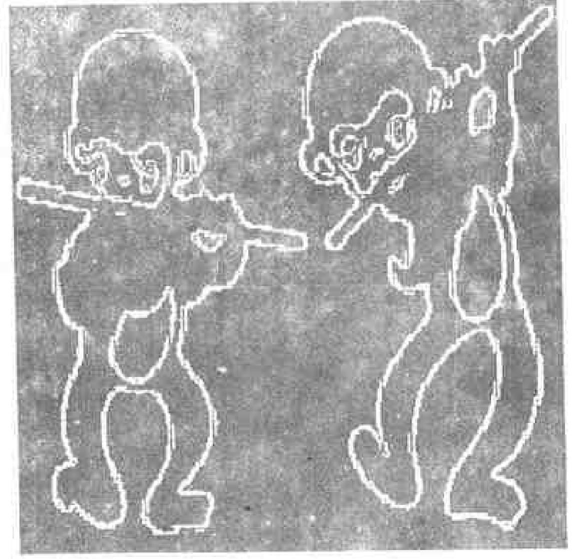


图 4—3—2 Roberts 算子边缘检测后的图像

§3.2 拉普拉斯算子边缘检测

一、功 能

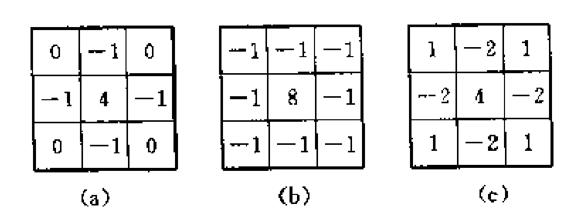
用拉普拉斯算子进行图像的边缘检测。

二、方法简介

函数 $f(x,y)$ 的拉普拉斯运算定义为

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

其中 ∇^2 称为拉普拉斯算子。下面给出常用的三种数字拉普拉斯算子，如图 4—3—3 所示。



0	−1	0
−1	4	−1
0	−1	0

(a)

−1	−1	−1
−1	8	−1
−1	−1	−1

(b)

1	−2	1
−2	4	−2
1	−2	1

(c)

图 4—3—3 拉普拉斯算子

三、使用说明

1. 子函数语句

void lapla(ip,jp,lx,ly)

2. **形参说明**

ip——无符号字符型二维数组，体积为lx×ly，输入图像，ip[i][j]表示在坐标(i,j)点处的像素值

jp——无符号字符型二维数组，体积为lx×ly，输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量，图像在x方向上的像素数。

ly——无符号长整型变量。图像在y方向上的像素数。

四、子函数程序(文件名:lapla.c)

```
#define ff(x,y) ip[lx*(y)+(x)]
#define gg(x,y) jp[lx*(y)+(x)]
void lapla(ip,jp,lx,ly)
unsigned char huge *ip,*jp;
unsigned long lx,ly;
{ int ix,iy,im,mx,my,jx,jy,jpv;
  static int lap[3][3][3] = {{0,-1,0,-1,4,-1,0,-1,0},
                             {-1,-1,-1,-1,8,-1,-1,-1,-1},
                             {1,-2,1,-2,4,-2,1,-2,1}};
  for (ix=0;ix<lx;ix++)
    { gg(ix,0) = 0;
      gg(ix,ly-1) = 0;
    }
  for (iy=1;iy<ly;iy++)
    { gg(0,iy) = 0;
      gg(lx-1,iy) = 0;
    }
  puts("Please choose mask (1,2 or 3):");
  printf("mask1:  0 -1  0  mask2:-1 -1 -1  mask3:  1 -2  1\n");
  printf("       -1  4 -1        -1  8 -1         -2  4 -2\n");
  printf("        0 -1  0        -1 -1 -1          1 -2  1\n");
  scanf("%d",&im);
  if ((im!=1)&&(im!=2)&&(im!=3)) return;
  im--;
  for (ix=1;ix<lx-1;ix++)
  for (iy=1;iy<ly-1;iy++)
    { jpv = 0;
      for (mx=0;mx<=2;mx++)
          jx = ix - 1 + mx;
```

```
            for (my=0;my<=2;my++)
              { jy = iy - 1 + my;
                jpv = jpv + ff(jx,jy) * lap[im][mx][my];
              }
          }
        gg(ix,iy) = abs(jpv);
      }
  }
```

五、例　题

用 Laplace 算子对猴子 “Monkey” 图像进行边缘检测，调用语句：

lapla(i-img,o-img,256l,256l);

输入图像：猴子 “Monkey” 图像(文件名：monkey.bmp)，图像大小为 256×256，见图 4—3—1。

输出图像：略。

§3.3　Sobel 算子边缘检测

一、功　能

用 Sobel 算子进行图像的边缘检测。

二、方法简介

设输入图像为 $f(x,y)$，输出图像为 $g(x,y)$，那么 Sobel 运算定义为

$$g(x,y) = |V_x| + |V_y|$$

其中

$$\begin{aligned} V_x = &\ [f(x-1,y-1) + 2f(x-1,y) + f(x-1,y+1)] \\ &- [f(x+1,y-1) + 2f(x+1,y) + f(x+1,y+1)] \\ V_y = &\ [f(x-1,y-1) + 2f(x,y-1) + f(x+1,y-1)] \\ &- [f(x-1,y+1) + 2f(x,y+1) + f(x+1,y+1)] \end{aligned}$$

三、使用说明

1. 子函数语句

void sobel(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)

点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:sobel.c)

```
#include "math.h"
#define ff(x,y) ip[lx * (y)+(x)]
#define gg(x,y) jp[lx * (y)+(x)]
void sobel(ip,jp,lx,ly)
unsigned char huge *ip, *jp;
unsigned long lx,ly;
{ int ix,iy,vx,vy;
  for (ix=0;ix<lx;ix++)
    { gg(ix,0) = 0;
      gg(ix,ly-1) = 0;
    }
  for (iy=1;iy<ly;iy++)
    { gg(0,iy) = 0;
      gg(lx-1,iy) = 0;
    }
  for (ix=1;ix<lx-1;ix++)
  for (iy=1;iy<ly-1;iy++)
    { vx = (ff(ix-1,iy-1) + 2 * ff(ix-1,iy) + ff(ix-1,iy+1) )
          -(ff(ix+1,iy-1) + 2 * ff(ix+1,iy) + ff(ix+1,iy+1) );
      vy = (ff(ix-1,iy-1) + 2 * ff(ix,iy-1) + ff(ix+1,iy-1) )
          -(ff(ix-1,iy+1) + 2 * ff(ix,iy+1) + ff(ix+1,iy+1) );
      gg(ix,iy) = abs(vx) + abs(vy);
    }
}
```

五、例　题

用 Sobel 算子对猴子"Monkey"图像进行边缘检测，调用语句:

```
sobel(i-img,o-img,256l,256l);
```

输入图像:猴子"Monkey"图像(文件名:monkey.bmp),图像大小为 256×256,见图 4-3-1。

输出图像:略。

§ 3.4　Robinson 算子边缘检测

一、功　能

用 Robinson 算子进行图像的边缘检测。

二、方法简介

Robinson 算子是一种边缘模板算子，它由理想的边缘子图像构成。依次用边缘模板去检测图像，与被检测区域最为相似的模板给出最大值。用该最大值作为算子的输出值，从而使边缘像素得到增强。

Robinson 算子由 8 个边缘模板组成，如图 4—3—4 所示。

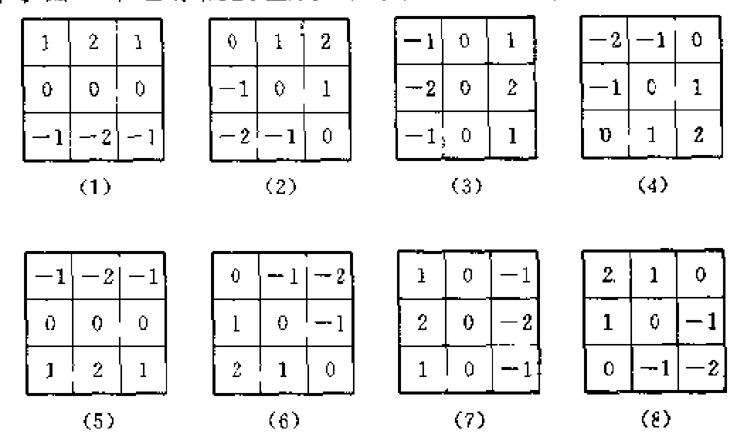


图 4—3—4　Robinson 算子

8 个算子模板对应的边缘方向如图 4—3—5 所示。

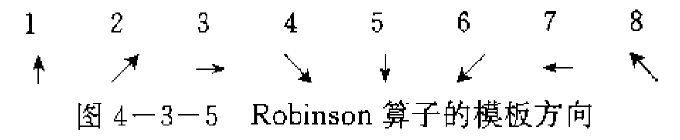


图 4—3—5　Robinson 算子的模板方向

三、使用说明

1. 子函数语句

void robinson(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:robinson.c)

```
#define ff(x,y) ip[lx * (y)+(x)]
#define gg(x,y) jp[lx * (y)+(x)]
void robinson(ip,jp,lx,ly)
unsigned char huge  * ip, * jp;
unsigned long lx,ly;
{ int ix,iy,im,mx,my,jx,jy,jpv,mxav;
  static int mask[8][3][3]=
    {   1,  2,  1,  0,  0,  0,-1,-2,-1,
        0,  1,  2,-1,  0,  1,-2,-1,  0,
       -1,  0,  1,-2,  0,  2,-1,  0,  1,
       -2,-1,  0,-1,  0,  1,  0,-1,  2,
       -1,-2,-1,  0,  0,  0,  1,  2,  1,
        0,-1,-2,  1,  0,-1,  2,  1,  0,
        1,  0,-1,  2,  0,  2,  1,  0,-1,
        2,  1,  0,  1,  0,-1,  0,-1,-2
    };
   for (ix=0;ix<lx;ix++)
     { gg(ix,0) = 0;
       gg(ix,ly-1) = 0;
     }
   for (iy=1;iy<ly;iy++)
     { gg(0,iy) = 0;
       gg(lx-1,iy) = 0;
     }
   for (ix=1;ix<lx-1;ix++)
   for (iy=1;iy<ly-1;iy++)
     { for (im=0;im<=7;im++)
          { jpv = 0;
            mxav = 0;
            for (mx=0;mx<=2;mx++)
              { jx = ix - 1 + mx;
                  for (my=0;my<=2;my++)
                  { jy = iy - 1 + my;
                    jpv = jpv + ff(jx,jy) * mask[im][mx][my];
                  }
              }
```

```
            if ((jpv>mxav)||(im==0)) mxav = jpv;
          }
        gg(ix,iy) = mxav;
      }
}
```

五、例　题

用 Robinson 算子对猴子“Monkey”图像进行边缘检测，调用语句：

robinson(i-img,o-img,256l,256l)；

输入图像：猴子“Monkey”图像(文件名：monkey. bmp)，图像大小为 256×256，见图 4－3－1。

输出图像：略。

§3.5　Kirsch 算子边缘检测

一、功　能

用 Kirsch 算子进行图像的边缘检测。

二、方法简介

Kirsch 算子是一种边缘模板算子，它由理想的边缘子图像构成。依次用边缘模板去检测图像，与被检测区域最为相似的模板给出最大值。用该最大值作为算子的输出值，从而使边缘像素得到增强。

Kirsch 算子由 8 个边缘模板组成，如图 4－3－6 所示。

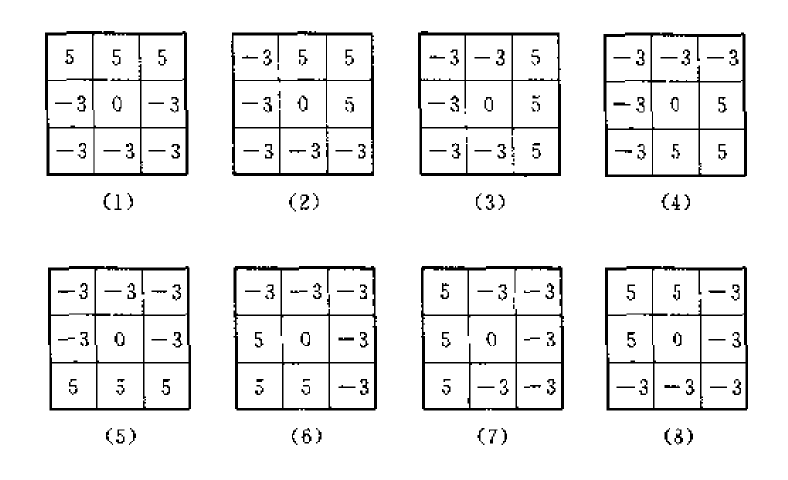


图 4－3－6　Kirsch 算子

8 个算子模板对应的边缘方向如图 4－3－7 所示。

1	2	3	4	5	6	7	8
↑	↗	→	↘	↓	↙	←	↖

图 4－3－7 Kirsch 算子的模板方向

三、使用说明

1. 子函数语句

void kirsch(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:kirsch.c)

```
#define ff(x,y) ip[lx * (y)+(x)]
#define gg(x,y) jp[lx * (y)+(x)]
void kirsch(ip,jp,lx,ly)
unsigned char huge  *ip, *jp;
unsigned long lx,ly;
{ int ix,iy,im,mx,my,jx,jy,jpv,mxav;
  static int mask[8][3][3]=
    {  5,  5,  5,-3,  0,-3,-3,-3,-3,
      -3,  5,  5,-3,  0,  5,-3,-3,-3,
      -3,-3,  5,-3,  0,  5,-3,-3,  5,
      -3,-3,-3,-3,  0,  5,-3,  5,  5,
      -3,-3,-3,-3,  0,-3,  5,  5,  5,
      -3,-3,-3,  5,  0,-3,  5,  5,-3
       5,-3,-3,  5,  0,-3,  5,-3,-3,
       5,  5,-3,  5,  0,-3,-3,-3,-3,
    };
  for (ix=0;ix<lx;ix++)
    { gg(ix,0) = 0;
      gg(ix,ly-1) = 0;
    }
  for (iy=1;iy<ly;iy++)
    { gg(0,iy) = 0;
```

```
      gg(lx-1,iy) = 0;
    }
  for (ix=1;ix<lx-1;ix++)
  for (iy=1;iy<ly-1;iy++)
    { for (im=0;im<=7;im++)
        { jpv = 0;
          for (mx=0;mx<=2;mx++)
            { jx = ix - 1 + mx;
              for (my=0;my<=2;my++)
                { jy = iy - 1 + my;
                  jpv = jpv + ff(jx,jy) * mask[im][mx][my];
                }
            }
          if ((jpv>mxav)||(im==0)) mxav = jpv;
        }
      gg(ix,iy) = mxav;
    }
}
```

五、例　题

用 Kirsch 算子对猴子"Monkey"图像进行边缘检测，调用语句：

kirsch(i-img,o-img,256l,256l);

输入图像：猴子"Monkey"图像(文件名：monkey.bmp)，图像大小为 256×256，见图 4－3－1。

输出图像：略。

§3.6 Prewitt 算子边缘检测

一、功　能

用 Prewitt 算子进行图像的边缘检测。

二、方法简介

Prewitt 算子是一种边缘模板算子，它由理想的边缘子图像构成。依次用边缘模板去检测图像，与被检测区域最为相似的模板给出最大值。用该最大值作为算子的输出值，从而使边缘像素得到增强。

Prewitt 算子由 8 个边缘模板组成，如图 4－3－8 所示。

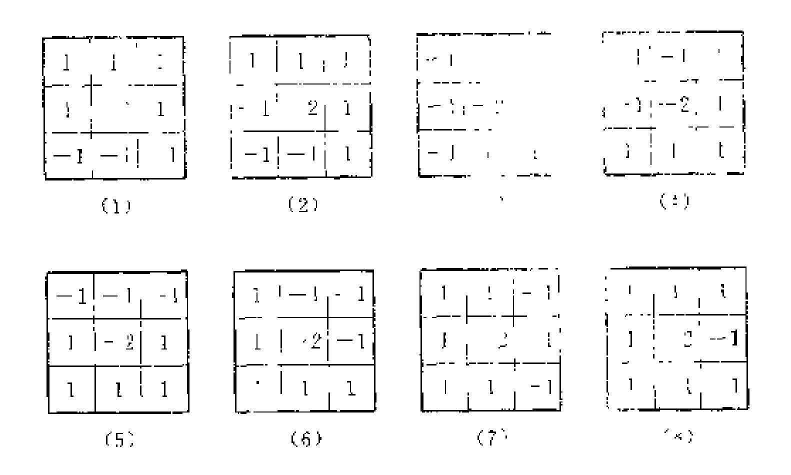


图4-3-8　Prewitt算子

8个算子模板对应的边缘方向如图4-3-9所示。

1	2	3	4	5	6	7	8
↑	↗	→	↘	↓	↙	←	↖

图4-3-9　Prewitt算子的模板方向

三、使用说明

1. 子函数语句

void prewitt(ip,jp,lx,ly)

2. 形参说明

ip——无符号字符型二维数组，体积为lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在x方向上的像素数。

ly——无符号长整型变量。图像在y方向上的像素数。

四、子函数程序(文件名:prewitt.c)

```
#define ff(x,y) ip[lx*(y)+(x)]
#define gg(x,y) jp[lx*(y)+(x)]
void prewitt(ip,jp,lx,ly)
unsigned char huge *ip,*jp;
unsigned long lx,ly;
{ int ix,iy,im,mx,my,jx,jy,jpv;
  int mxav,mask[8][3][3]=
    { 1, 1, 1, 1,-2, 1,-1,-1,-1,
      1, 1, 1,-1,-2, 1,-1,-1, 1,
     -1, 1, 1,-1,-2, 1,-1, 1, 1,
```

```
    -1,-1,  1,-1,-2,  1,  1,  1,  1,
    -1,-1,-1,  1,-2,  1,  1,  1,  1,
     1,-1,-1,  1,-2,-1,  1,  1,  1,
     1,  1,-1,  1,-2,-1,  1,  1,-1,
     1,  1,  1,  1,-2,-1,  1,-1,-1,
  };
 for (ix=0;ix<lx;ix++)
   { gg(ix,0) = 0;
     gg(ix,ly-1) = 0;
   }
 for (iy=1;iy<ly;iy++)
   { gg(0,iy) = 0;
     gg(lx-1,iy) = 0;
   }
 for (ix=1;ix<lx-1;ix++)
 for (iy=1;iy<ly-1;iy++)
   { for (im=0;im<=7;im++)
       { jpv = 0;
         mxav = 0;
         for (mx=0;mx<=2;mx++)
           { jx = ix - 1 + mx;
             for (my=0;my<=2;my++)
               { jy = iy - 1 + my;
                 jpv = jpv + ff(jx,jy) * mask[im][mx][my];
               }
           }
         if ((jpv>mxav)||(im==0)) mxav = jpv;
         }
     gg(ix,iy) = mxav;
   }
}
```

五、例　题

用 Prewitt 算子对猴子“Monkey”图像进行边缘检测，调用语句：

```
prewitt(i-img,o-img,256l,256l);
```

输入图像:猴子“Monkey”图像(文件名:monkey.bmp),图像大小为 256×256,见图 4-3-1。

输出图像:略。

第四章　图像细化

§4.1　Hilditch 细化算法

一、功　能

用 Hilditch 算法进行图像细化。

二、方法简介

图像细化就是把二值图像中具有一定宽度的线条状区域变成一条薄线(即只有一个像素宽度)。图像细化大大地压缩了原始图像的数据量，并保持其形状的基本拓扑结构不变，从而为文字识别中的特征抽取等应用奠定了基础。细化算法应满足以下条件：(1) 将条形区域变成一条薄线；(2) 薄线应位于原条形区域的中心；(3) 薄线应保持原图像的拓扑特性。

Hilditch 算法适用于输入图像为 0 和 1 的二值图像。像素值为 1 的区域是需要细化的部分，像素值为 0 的区域是背景。Hilditch 细化算法可描述如下：

设 p 为被检测的像素，$f(p)$为像素 p 的灰度值，$n_i(i=1,2,\cdots,8)$为 p 的 8 邻域像素，n_i 的位置如图 4—4—1 所示。

n_4	n_3	n_2
n_5	p	n_1
n_6	n_7	n_8

图 4—4—1　像素 p 的 8 邻域

设集合 $I=\{1\}$ 表示需要细化的像素子集，集合 $N=\{g\mid g-m\leqslant 0\}$ 表示背景像素子集，集合 $R=\{-m\}$ 表示在第 m 次减薄时，I 中被减掉的像素。

图像细化的减薄条件为：

1. $f(p)\in I$

2. $U(p)\geqslant 1$，其中 $U(p)=a_1+a_3+a_5+a_7$，这里 a_i 为

$$a_i=\begin{cases}1 & ,\ f(n_i)\in N\\ 0 & ,\ \text{其他}\end{cases}$$

3. $V(p)\geqslant 2$，其中 $V(p)=\sum_{i=1}^{8}(1-a_i)$

4. $W(p)\geqslant 1$，其中 $W(p)=\sum_{i=1}^{8}c_i$，这里 c_i 为

$$c_i=\begin{cases}1 & ,\ f(n_i)\in I\\ 0 & ,\ \text{其他}\end{cases}$$

5. $x(p)=1$，其中 $x(p)=\sum_{i=1}^{1} b_i$，这里 b_i 为

$$b_i=\begin{cases}1 & ，f(n_{2i-1})\in N \text{ 并且 } f(n_{2i})\in I\cup R \text{ 或 } f(n_{2i-1})\in I\cup R \\ 0 & ，\text{其他}\end{cases}$$

6. $f(n_i)\notin R$ 或 $x_i(p)=1$，$(i=3,5)$，其中 $x_i(p)$ 表示对 p 的第 i 个邻域像素的 $x(p)$。

三、使用说明

1. 子函数语句

void hilditch(io-image,lx,ly)

注意：子函数 hilditch()适用于灰度值为 0 和 1 的二值图像的细化。当对灰度值为 0 和 255 的二值图像进行细化时，还要用到以下两个子函数：

void beforethin(ip,jp,lx,ly)

void afterthin(jp,lx,ly)

其中子函数 beforethin()用于把灰度值为 0 和 255 的二值图像转换成灰度值为 0 和 1 的二值图像，子函数 afterthin()用于把灰度值为 0 和 1 的二值图像转换成灰度值为 0 和 255 的二值图像。

2. 形参说明

io-image——无符号字符型二维数组，体积为 lx×ly。io-image[i][j]表示在坐标(i,j)点处的像素值。开始时存放输入图像，最后存放输出图像。

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:hilditch.c)

```
#include "alloc.h"
void hilditch(io-image,lx,ly)
void huge *io-image;
unsigned long lx,ly;
{ char huge *f, *g;
  char n[10];
  unsigned int counter;
  short k,shori,xx,nrn;
  long i,j,kk,kk11,kk12,kk13,kk21,kk22,kk23,kk31,kk32,kk33,size;
  size = (long)lx * (long)ly;
```

```
g = (unsigned char huge * )farmalloc(size);
if (g == NULL)
  { printf("memory allocate error ! \n");
    exit(-1);
  }
f = (unsigned char huge * )io-image;
for (i=0;i<lx;i++)
for (j=0;j<ly;j++)
  { kk = i * ly+j;
    if (f[kk] != 0)
         { f[kk] = 1;
           g[kk] = f[kk];
         }
  }
counter = 1;
do
  { printf(" %4d * ",counter);
    counter++;
    if (bioskey(1)) break;
    shori = 0;
    for (i=0;i<lx;i++)
    for (j=0;j<ly;j++)
       { kk = i * ly+j;
         if (f[kk]<0) f[kk] = 0;
         g[kk] = f[kk];
       }
    for (i=1;i<lx-1;i++)
    for (j=1;j<ly-1;j++)
       { kk = i * ly+j;
         if (f[kk] != 1) continue;
         kk11 = (i-1) * ly + j - 1;
         kk12 = kk11 + 1;
         kk13 = kk12 + 1;
         kk21 = (i) * ly + j - 1;
         kk22 = kk21 + 1;
         kk23 = kk22 + 1;
         kk31 = (i+1) * ly + j - 1;
         kk32 = kk31 + 1;
```

```
kk33 = kk32 + 1;
if ((g[kk12] && g[kk21] && g[kk23] && g[kk32]) != 0)
  { continue; }
nrn = g[kk11] + g[kk12] + g[kk13] + g[kk21] + g[kk23]
      + g[kk31] + g[kk32] + g[kk33];
if (nrn <= 1)
  { f[kk22] = 2;
    continue;
  }
n[4] = f[kk11]; n[3] = f[kk12]; n[2] = f[kk13];
n[5] = f[kk21]; n[1] = f[kk23];
n[6] = f[kk31]; n[7] = f[kk32]; n[8] = f[kk33];
n[9] = n[1];
xx = 0;
for (k=1;k<8;k=k+2)
  { if ((!n[k]) && (n[k+1] || n[k+2])) xx++; }
if (xx != 1)
  { f[kk22] = 2;
    continue;
  }
if (f[kk12] == -1)
  { f[kk12] = 0;
    n[3] = 0;
    xx = 0;
    for (k=1;k<8;k=k+2)
      { if ((!n[k]) && (n[k+1] || n[k+2])) xx++; }
    if (xx != 1)
      { f[kk12] = -1;
        continue;
      }
    f[kk12] = -1;
    n[3] = -1;
  }
  if (f[kk21] != -1)
    { f[kk22] = -1;
      shori = 1;
      continue;
    }
```

```
            f[kk21] = 0;
            n[5] = 0;
            xx = 0;
            for (k=1;k<8;k=k+2)
            { if ((!n[k]) && (n[k+1] || n[k+2])) xx++; }
            if (xx == 1)
              { f[kk21] = -1;
              f[kk22] = -1;
              shori = 1;
              }
            else
              { f[kk21] = -1; }
        }
    } while ( shori );
  farfree(g);
}

void beforethin(ip,jp,lx,ly)
unsigned char huge * ip, * jp;
unsigned long lx,ly;
{ unsigned long i,j;
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { if (ip[i * lx+j]>0)
        { jp[i * lx+j] = 0; }
      else
        { jp[i * lx+j] = 1; }
    }
}

void afterthin(jp,lx,ly)
unsigned char huge *jp;
unsigned long lx,ly;
{ unsigned long i,j;
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { jp[i * lx+j] = (1 - jp[i * lx+j]) * 255; }
}
```

五、例　题

用 Hildiltch 算法对数字“8”图像进行细化，调用语句：

```
beforethin(i-img,o-img,256l,256l);
hilditch(o-img,256l,256l);
afterthin(o-img,256l,256l);
```

输入图像：数字“8”图像（文件名：8.bmp），图像大小为 256×256，见图 4—4—2。

输出图像：见图 4—4—3。

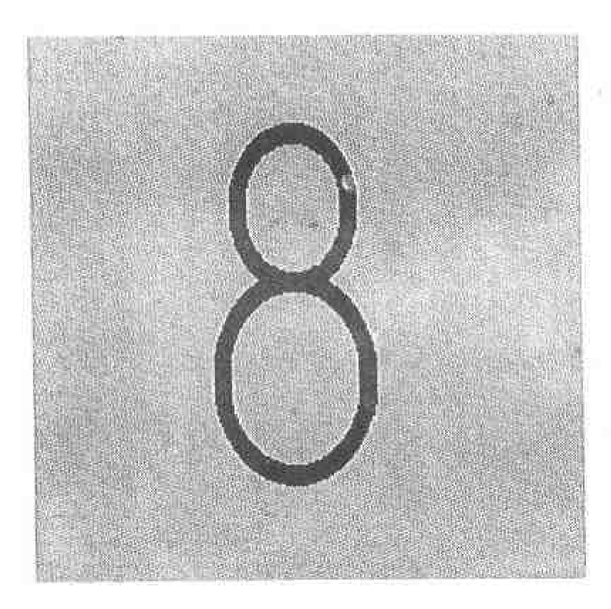


图 4—4—2　数字“8”图像

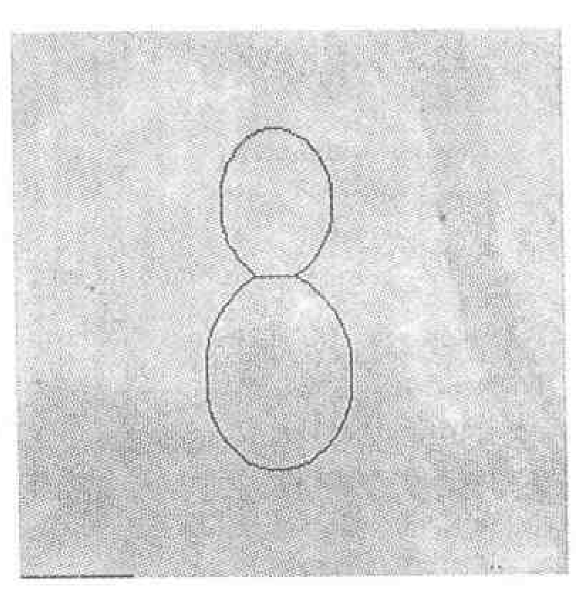


图 4—4—3　细化后的图像

§4.2　Pavlidis 细化算法

一、功　能

用 Pavlidis 算法进行图像细化。

二、方法简介

Pavlidis 的异步细化算法是通过并行与串行混合处理来实现细化，它用位运算进行特定模式的匹配，所得的骨架形态是 8-连接的。该算法适用于像素值为 0 和 1 的二值图像。Pavlidis 算法的细节见文献[58]。

三、使用说明

1. 子函数语句

void pavlidis(io-image,lx,ly)

注意：子函数 pavlidis()适用于灰度值为 0 和 1 的二值图像的细化。当对灰度值为 0 和 255 的二值图像进行细化时，还要用到以下两个子函数：

void beforethin(ip,jp,lx,ly)

void afterthin(jp,lx,ly)

其中子函数 beforethin()用于把灰度值为 0 和 255 的二值图像转换成灰度值为 0 和 1 的二值图像，子函数 afterthin()用于把灰度值为 0 和 1 的二值图像转换成灰度值为 0 和 255 的二值图像。

2. 形参说明

io-image——无符号字符型二维数组，体积为 lx×ly。io-image[i][j]表示在坐标(i,j)点处的像素值。开始时存放输入图像，最后存放输出图像。

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:pavlidis.c)

```
void pavlidis(io-image,lx,ly)
void huge *io-image;
unsigned long lx,ly;
{ char erase, n[8];
  char huge *f;
  unsigned char bdr1,bdr2,bdr4,bdr5;
  short c,k,b;
  long i,j,kk,kk1,kk2,kk3;
  f = (unsigned char huge *)io-image;
  for (i=1;i<lx-1;i++)
  for (j=1;j<ly-1;j++)
    { kk = i*ly + j;
      if ( f[kk] ) f[kk] = 1;
    }
  for (i=0,kk1=0,kk2=ly-1;  i<lx;  i++,kk1+=ly,kk2+=ly)
    { f[kk1] = 0;
      f[kk2] = 0;
    }
  for (j=0,kk=(lx-1)*ly;j<ly;j++,kk++)
```

```
      { f[j] = 0;
        f[kk] = 0;
      }
   c = 5;
   erase = 1;
   while ( erase )
      { c++;
        for (i=1;i<lx-1;i++)
        for (j=1;j<ly-1;j++)
          { kk = i * ly + j;
            if (f[kk] != 1) continue;
            kk1 = kk - ly - 1;
              kk2 = kk1 + 1;
              kk3 = kk2 + 1;
              n[3] = f[kk1];
              n[2] = f[kk2];
              n[1] = f[kk3];
              kk1 = kk - 1;
              kk3 = kk + 1;
              n[4] = f[kk1];
              n[0] = f[kk3];
            kk1 = kk + ly - 1;
              kk2 = kk1 + 1;
              kk3 = kk2 + 1;
              n[5] = f[kk1];
              n[6] = f[kk2];
              n[7] = f[kk3];
            bdr1 = 0;
            for (k=0;k<8;k++)
              { if (n[k]>=1) bdr1 |= 0x80>>k; }
            if ((bdr1 & 0252) == 0252) continue;
            f[kk] = 2;
            b = 0;
            for (k=0;k<=7;k++)
              { b += bdr1 & (0x80>>k); }
            if (b <= 1) f[kk] = 3;
            if ((bdr1 & 0160)!=0 && (bdr1 & 07)!=0 && (bdr1 & 0210)==0)
              { f[kk] = 3; }
```

```
        else if ((bdr1&0301)!=0 &&(bdr1&034)!=0 &&(bdr1&042)==0)
          { f[kk] = 3; }
        else if ((bdr1 & 0202) == 0 && (bdr1 & 01) != 0)
          { f[kk] = 3; }
        else if ((bdr1 & 0240) == 0 && (bdr1 & 0100) ! = 0)
          { f[kk] = 3; }
        else if ((bdr1 & 050) == 0 && (bdr1 & 020) ! = 0)
          { f[kk] = 3; }
        else if((bdr1 & 012) == 0 && (bdr1 & 04) ! = 0)
          { f[kk] = 3; }
      }
  for (i=1;i<lx-1;i++)
  for (j=1;j<ly-1;j++)
    { kk = i * ly + j;
      if ( ! f[kk] ) continue;
      kk1 = kk - ly - 1;
      kk2 = kk1 + 1;
      kk3 = kk2 + 1;
      n[3] = f[kk1];
      n[2] = f[kk2];
      n[1] = f[kk3];
      kk1 = kk - 1;
      kk3 = kk + 1;
      n[4] = f[kk1];
      n[0] = f[kk3];
      kk1 = kk + ly - 1;
      kk2 = kk1 + 1;
      kk3 = kk2 + 1;
      n[5] = f[kk1];
      n[6] = f[kk2];
      n[7] = f[kk3];
      bdr1 = bdr2 = 0;
      for (k=0;k<=7;k++)
        { if (n[k] >= 1) bdr1 |= 0x80>>k;
          if (n[k] >= 2) bdr2 |= 0x80>>k;
        }
      if (bdr1 == bdr2)
        { f[kk] = 4;
```

```
        continue;
      }
    if (f[kk] != 2) continue;
    if ( (bdr2 & 0200) != 0 && (bdr1 & 010) == 0 &&
        ((bdr1 & 0100) != 0 && (bdr1 & 001) != 0 ||
        ((bdr1 & 0100) != 0 || (bdr1 & 001) != 0) &&
        (bdr1 & 060) != 0 && (bdr1 & 06) != 0) )
      { f[kk] = 4; }
    else if ((bdr2 & 040) != 0 && (bdr1 & 02) == 0 &&
             ((bdr1 & 020) != 0 && (bdr1 & 0100) != 0 ||
             ((bdr1 & 020) != 0 || (bdr1 & 0100) != 0) &&
             (bdr1 & 014) != 0 && (bdr1 & 0201) != 0))
      { f[kk] = 4; }
    else if ((bdr2 & 010) != 0 && (bdr1 & 0200) == 0 &&
             ((bdr1 & 04) != 0 && (bdr1 & 020) != 0 ||
             ((bdr1 & 04) != 0 || (bdr1 & 020) != 0) &&
             (bdr1 & 03) != 0 && (bdr1 & 0140) != 0))
      { f[kk] = 4; }
    else if ((bdr2 & 02) != 0 && ( ! (bdr1 & 040)) &&
             ((bdr1 & 01) != 0 && (bdr1 & 04) != 0 ||
             ((bdr1 & 01) != 0 || (bdr1 & 04) != 0) &&
             (bdr1 & 0300) != 0 && (bdr1 & 030) != 0))
      { f[kk] = 4; }
  }
for (i=1;i<lx-1;i++)
for (j=1;j<ly-1;j++)
  { kk = i * ly + j;
    if (f[kk] != 2) continue;
    kk1 = kk - ly - 1;
    kk2 = kk1 + 1;
    kk3 = kk2 + 1;
    n[3] = f[kk1];
    n[2] = f[kk2];
    n[1] = f[kk3];
    kk1 = kk - 1;
    kk3 = kk + 1;
    n[4] = f[kk1];
    n[0] = f[kk3];
```

```
        kk1 = kk + ly - 1;
        kk2 = kk1 + 1;
        kk3 = kk2 + 1;
        n[5] = f[kk1];
        n[6] = f[kk2];
        n[7] = f[kk3];
        bdr4 = bdr5 = 0;
        for (k=0;k<=7;k++)
          { if (n[k] >= 4) bdr4 |= 0x80>>k;
            if (n[k] >= 5) bdr5 |= 0x80>>k;
          }
        if ((bdr4 & 010) == 0)
          { f[kk] = 5;
              continue;
          }
        if ((bdr4 & 040) == 0 && bdr5 == 0)
          { f[kk] = 5;
              continue;
          }
        if (f[kk]==3 || f[kk]==4) f[kk]=c;
      }
    erase = 0;
    for (i=1;i<lx-1;i++)
    for (j=1;j<ly-1;j++)
      { kk = i * ly + j;
        if (f[kk]==2 || f[kk]==5)
          { erase = 1;
            f[kk] = 0;
          }
      }
  }
}

void beforethin(ip,jp,lx,ly)
unsigned char huge * ip, * jp;
unsigned long lx,ly;
{ unsigned long i,j;
  for (i=0;i<ly;i++)
```

```
  for (j=0;j<lx;j++)
    { if (ip[i * lx+j]>0)
        { jp[i * lx+j] = 0; }
      else
        { jp[i * lx+j] = 1; }
    }
}

void afterthin(jp,lx,ly)
unsigned char huge  * jp;
unsigned long lx,ly;
{ unsigned long i,j;
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { jp[i * lx+j] = (1 - jp[i * lx+j]) * 255; }
}
```

五、例　题

用 Pavlidis 算法对数字“8”图像进行细化，调用语句：

```
beforethin(i-img,o-img,256l,256l);
pavlidis(o-img,256l,256l);
afterthin(o-img,256l,256l);
```

输入图像：数字“8”图像（文件名：8.bmp），图像大小为 256×256，见图 4—4—2。

输出图像：略。

§4.3　Rosenfeld 细化算法

一、功　能

用 Rosenfeld 算法进行图像细化。

二、方法简介

Rosenfeld 算法是一种并行细化算法，所得的骨架形态是 8-连接的。该算法适用于像素值为 0 和 1 的二值图像。有关算法的细节见文献[59]。

三、使用说明

1. 子函数语句

```
void rosenfd(io-image,lx,ly)
```

注意：子函数 rosenfd()适用于灰度值为 0 和 1 的二值图像的细化。当对灰度值为 0 和 255 的二值图像进行细化时，还要用到以下两个子函数：

void beforethin(ip,jp,lx,ly)

void afterthin(jp,lx,ly)

其中子函数 beforethin()用于把灰度值为 0 和 255 的二值图像转换成灰度值为 0 和 1 的二值图像，子函数 afterthin()用于把灰度值为 0 和 1 的二值图像转换成灰度值为 0 和 255 的二值图像。

2. 形参说明

io-image—— 无符号字符型二维数组，体积为 lx×ly。io-image[i][j]表示在坐标(i,j)点处的像素值。开始时存放输入图像，最后存放输出图像。

ip——无符号字符型二维数组，体积为 lx×ly。输入图像，ip[i][j]表示在坐标(i,j)点处的像素值。

jp——无符号字符型二维数组，体积为 lx×ly。输出图像，jp[i][j]表示在坐标(i,j)点处的像素值。

lx——无符号长整型变量。图像在 x 方向上的像素数。

ly——无符号长整型变量。图像在 y 方向上的像素数。

四、子函数程序(文件名:rosenfd.c)

```
void rosenfd(io-image,lx,ly)
void huge *io-image;
unsigned long lx,ly;
{ char huge *f, *g;
  char n[10];
  char a[5] = {0, -1, 1, 0, 0}, b[5] = {0, 0, 0, 1, -1};
  char nrnd,cond,n48,n26,n24,n46,n68,n82,n123,n345,n567,n781;
  short k,shori;
  long i,j,ii,jj,kk,kk1,kk2,kk3,size;
  size = (long)lx * (long)ly;
  g = (unsigned char huge *)farmalloc(size);
  if ( g == NULL)
    { printf("memory allocate error ! \n");
      exit(-1);
    }
  f = (unsigned char huge *)io-image;
  for (kk=0l;kk<size;kk++)
    { g[kk] = f[kk]; }
  do
    { shori = 0;
```

```
for (k=1;k<=4;k++)
  { for (i=1;i<lx-1;i++)
      { ii = i + a[k];
        for (j=1;j<ly-1;j++)
          { kk = i * ly + j;
            if ( ! f[kk] ) continue;
            jj = j + b[k];
            kk1 = ii * ly + jj;
            if (f[kk1]) continue;
            kk1 = kk - ly - 1;
            kk2 = kk1 + 1;
            kk3 = kk2 + 1;
            n[3] = f[kk1];
            n[2] = f[kk2];
            n[1] = f[kk3];
            kk1 = kk - 1;
            kk3 = kk + 1;
            n[4] = f[kk1];
            n[8] = f[kk3];
            kk1 = kk + ly - 1;
            kk2 = kk1 + 1;
            kk3 = kk2 + 1;
            n[5] = f[kk1];
            n[6] = f[kk2];
            n[7] = f[kk3];
            nrnd = n[1] + n[2] + n[3] + n[4]
                   +n[5] + n[6] + n[7] + n[8];
            if (nrnd <= 1) continue;
            cond = 0;
            n48 = n[4] + n[8];
            n26 = n[2] + n[6];
            n24 = n[2] + n[4];
            n46 = n[4] + n[6];
            n68 = n[6] + n[8];
            n82 = n[8] + n[2];
            n123 = n[1] + n[2] + n[3];
            n345 = n[3] + n[4] + n[5];
            n567 = n[5] + n[6] + n[7];
```

```
n781 = n[7] + n[8] + n[1];
if (n[2] == 1 && n48 == 0 && n567 > 0)
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[6] == 1 && n48 == 0 && n123 > 0)
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[8] == 1 && n26 == 0 && n345 > 0)
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[4] == 1 && n26 == 0 && n781 > 0)
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[5] == 1 && n46 == 0 )
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[7] == 1 && n68 == 0 )
  { if ( ! cond ) continue;
    g[kk] = 0;
    shori = 1;
    continue;
  }
if (n[1] == 1 && n82 == 0 )
```

```
                { if ( ! cond ) continue;
                  g[kk] = 0;
                  shori = 1;
                  continue;
                }
              if (n[3] == 1 && n24 == 0 )
                { if ( ! cond) continue;
                  g[kk] = 0;
                  shori = 1;
                  continue;
                }
              cond = 1;
              if ( ! cond ) continue;
              g[kk] = 0;
              shori = 1;
            }
          }
        for (i=0;i<lx;i++)
        for (j=0;j<ly;j++)
          { kk = i*ly + j;
            f[kk] = g[kk];
          }
      }
    } while ( shori );
  farfree(g);
}

void beforethin(ip,jp,lx,ly)
unsigned char huge * ip, * jp;
unsigned long lx,ly;
{ unsigned long i,j;
   for (i=0;i<ly;i++)
   for (j=0;j<lx;j++)
     { if (ip[i*lx+j]>0)
          { jp[i*lx+j] = 0; }
       else
          { jp[i*lx+j] = 1; }
     }
```

```
}

void afterthin(jp,lx,ly)
unsigned char huge *jp;
unsigned long lx,ly;
{ unsigned long i,j;
  for (i=0;i<ly;i++)
  for (j=0;j<lx;j++)
    { jp[i*lx+j] = (1 - jp[i*lx+j])*255; }
}
```

五、例 题

用 Rosenfeld 算法对数字"8"图像进行细化，调用语句：

```
beforethin(i-img,o-img,256l,256l);
rosenfd(o-img,256l,256l);
afterthin(o-img,256l,256l);
```

输入图像：数字"8"图像(文件名：8.bmp)，图像大小为 256×256，见图 4—4—2。

输出图像：略。

第五篇　人工神经网络

第一章　神经网络模型

§1.1　多层感知器神经网络

一、功　能

多层感知器神经网络模型与误差反向传播学习算法。

二、方法简介

多层感知器神经网络由输入层、输出层和若干个隐层组成。输入层有 N 个节点，输出层有 M 个节点，每个隐层有若干节点。第 p 个训练样本为 $X_p=[x_{p1}, x_{p2}, \cdots, x_{pN}]$，对应的理想输出为 $D_p=[d_{p1}, d_{p2}, \cdots, d_{pM}]$。网络中当前层第 j 个神经元为 u_j，其输入 s_{pj}为

$$s_{pj} = \sum_i w_{ji} o_{pi} + \theta_j$$

其中 o_{pi}是上一层第 i 个神经元的输出，w_{ji}是上一层第 i 个神经元的输出与本层第 j 个神经元的输入之间的连接权，θ_j是本层第 j 个神经元的阀值。神经元 u_j 的输出 o_{pj}为

$$o_{pj} = f(s_{pj})$$

这里 $f(\cdot)$是非线性 S 型函数

$$f(x) = \frac{1}{1+\mathrm{e}^{-x}}$$

多层感知器神经网络的误差反向传播学习算法如下：

1. 置各权和阀值的初始值 $w_{ji}(0)$、$\theta_j(0)$ 为小的随机数。
2. 给定 P 个训练样本 $X_p(p=1,2,\cdots,P)$ 和对应的理想输出 $D_p(p=1,2,\cdots,P)$ 。
3. 信息前向传送：

   计算网络各层的输出

$$o_{pj} = f(s_{pj}) = f(\sum_i w_{ji} o_{pi} + \theta_j)$$

4. 误差反向传播：

隐层：　$$\delta_{pj} = o_{pj}(1-o_{pj}) \sum_k \delta_{pk} w_{jk}$$

输出层：　$$\delta_{pj} = o_{pj}(1-o_{pj})(d_{pj}-o_{pj})$$

5. 修改权和阀值

$$w_{ji}(t+1) = w_{ji}(t) + \mu \delta_{pj} o_{pi}$$
$$\theta_j(t+1) = \theta_j(t) + \mu \delta_{pj}$$

其中 μ 是学习率。

6. 重复 2～5 步，直至 P 个样本都训练一遍。

7. 判断是否满足精度要求。若满足，则停止训练；否则，转至第 2 步。

三、使用说明

1. 子函数语句

信息前向传送：

void feedforward(layernum,n,k,y,w,b)

误差反向传播(计算 δ_{pj})：

void calcdelta(layernum,n,k,d,y,w,delta)

更新连接权值 w_{ji} 和阀值 θ_j：

void update(layernum,n,k,y,delta,miu,w,b)

计算 S 型函数的值：

double f(x)

计算 S 型函数的导数值：

double fd(x)

初始化权值 w_{ji} 和阀值 θ_j：

void init(layernum,n,k,w,b)

输入网络的输入模式：

void input(n,y)

输出(显示)网络的结果：

void output(layernum,n,k,y)

存储网络的各种参数：

void saveparameter(layernum,n,k,w,b)

读取网络的各种参数：

void readparameter(layernum,n,k,w,b)

2. 形参说明

layernum——整型变量。多层感知器神经网络的层数。

n——整型一维数组，长度为 layernum。n[i]表示神经网络第 i 层的节点数。

k——整型变量。表示数组 n[i](i=0,1,…,layernum－1)的最大值。

w——双精度实型三维数组，体积为 layernumn×k×k。w[l][i][j]表示第 l－1 层第 i 个节点与第 l 层第 j 个节点之间的连接权值。

b——双精度实型二维数组，体积为 layernum×k。b[l][j]表示第 l 层第 j 个神经元的阀值。

delta——双精度实型二维数组，体积为 layernum×k。delta[l][j]表示第 l 层第 j 个神经元的误差信号 δ_{pj}。

y——双精度实型二维数组，体积为 layernum×k。y[l][j]表示第 l 层第 j 个神经元的输出值。

d——双精度实型一维数组，长度为 n[layernum−1]。d[j]表示输出层第 j 个神经元的理想输出。

miu——双精度实型变量。神经网络的学习率 μ。

四、子函数程序(文件名:mlp.c)

```
#include "stdio.h"
#include "math.h"
#include "uniform.c"
static void feedforward(layernum,n,k,y,w,b)
int k,layernum,n[];
double w[],y[],b[];
{ int i,j,l;
  double u;
  double f();
  for (l=1;l<layernum;l++)
    { for (j=0;j<n[l];j++)
        { u=0.0;
          for (i=0;i<n[l-1];i++)
            { u = u + w[(l*k+i)*k+j]*y[(l-1)*k+i]; }
          u = u + b[l*k-j];
          y[l*k+j] = f(u);
        }
    }
}

static void calcdelta(layernum,n,k,d,y,w,delta)
int k,layernum,n[];
double d[],w[],y[],delta[];
{ int i,j,l,l1;
  double fd();
  for (j=0;j<n[layernum-1];j++)
    { l1 = layernum - 1;
      delta[l1*k+j] = (d[j]-y[l1*k+j])*fd(y[l1*k+j]);
    }
  for (l=layernum-2;l>0;l--)
    { for (j=0;j<n[l];j++)
```

```
            { delta[l * k+j] = 0.0;
              for (i=0;i<n[l+1];i++)
              { delta[l * k+j] += delta[(l+1) * k+i] * w[((l+1) * k+j) * k+
i];}
              delta[l * k+j] = fd(y[l * k+j]) * delta[l * k+j];
            }
        }
    }

    static void update(layernum,n,k,y,delta,miu,w,b)
    int k,layernum,n[];
    double miu,b[],w[],y[],delta[];
    { int i,j,l;
      for (l=1;l<layernum;l++)
        { for (j=0;j<n[l];j++)
            { for (i=0;i<n[l-1];i++)
              { w[(l * k+i) * k+j] += miu * delta[l * k+j] * y[(l-1) * k+i]; }
              b[l * k+j] += miu * delta[l * k+j];
            }
        }
    }

    static double f(x)
    double x;
    { double f1;
      f1 = 1.0/(1.0+exp(-x));
      return(f1);
    }

    static double fd(x)
    double x;
    { double fd1;
      fd1 = x * (1-x);
      return(fd1);
    }

    static void init(layernum,n,k,w,b)
    int k,layernum,n[];
```

```
double w[],b[];
{ int i,j,l;
  long int seed;
  double uniform();
  seed = 13579l;
  for (l=1;l<layernum;l++)
    { for (j=0;j<n[l];j++)
        { for (i=0;i<n[l-1];i++)
            { w[(l*k+i)*k+j] = uniform(0.0,1.0,&seed); }
          b[l*k+j] = uniform(0.0,1.0,&seed);
        }
    }
}

static void input(n,y)
int n[];
double y[];
{ int j;
  printf("\nInput of MLP");
  for (j=0;j<n[0];j++)
    { printf("\n\nInput pattern Y(0,%d) = ",j);
      scanf("%lf",&y[j]);
    }
}

static void output(layernum,n,k,y)
int k,layernum,n[];
double y[];
{ int j;
  printf("\nOutput of MLP\n");
  for (j=0;j<n[layernum-1];j++)
    { printf("\nY(%d,%d) = %lf",layernum-1,j,y[(layernum-1)*k+j]);}
}

static void saveparameter(layernum,n,k,w,b)
int k,layernum,n[];
double w[],b[];
{ int i,j,l;
```

```
  char fname1[14];
  FILE *fp;
  puts("\nInput name of network data file");
  scanf("%s",fname1);
  if ((fp = fopen(fname1,"w")) == NULL)
    { puts("\nCannot open file");
      exit(0);
    }
  fprintf(fp,"%d\n",layernum);
  for (l=0;l<layernum;l++)
    { fprintf(fp,"%d\n",n[l]); }
  for (l=1;l<layernum;l++)
    { for (j=0;j<n[l];j++)
        { fprintf(fp,"%lf\n",b[l*k+j]);
          for (i=0;i<n[l-1];i++)
            { fprintf(fp,"%lf\n",w[(l*k+i)*k+j]); }
        }
    }
  fclose(fp);
}

static void readparameter(layernum,n,k,w,b)
int *k, *layernum,n[];
double w[],b[];
{ int i,j,l;
  char fname2[14];
  FILE *fp;
  puts("\nInput the name of trained network data file");
  scanf("%s",fname2);
  if ((fp = fopen(fname2,"r")) == NULL)
    { puts("\n\7Cannot open file");
      exit(0);
    }
  fscanf(fp,"%d",layernum);
  for (l=0;l<(*layernum);l+-)
    { fscanf(fp,"%d",&n[l]); }
  *k = n[0];
  for (l=1;l<(*layernum);l++)
```

```
    { if ( *k < n[l]) *k = n[l]; }
  for (l=1;l<( *layernum);l++)
    { for (j=0;j<n[l];j++)
        { fscanf(fp," %lf",&b[l * ( *k)+j]);
          for (i=0;i<n[l-1];i++)
            { fscanf(fp," %lf",&w[(l * ( *k)+i) * ( *k)+j]); }
        }
    }
  fclose(fp);
}
```

五、例　题

用多层感知器神经网络求解异或判决问题。异或问题可描述如下

x_1	x_2	d
0	0	0
0	1	1
1	0	1
1	1	0

其中 x_1 和 x_2 是输入，d 是理想输出。多层感知器的参数为：输入层 2 个神经元，输出层 1 个神经元，隐层 6 个神经元，学习率 $\mu=0.5$，训练次数为 5000，精度 eps＝0.0001，训练样本数为 4，训练样本和对应的理想输出存储在数据文件 xor.dat 中，其存储格式为 x_1，x_2，d，即 0，0，0，0，1，1，1，0，1，1，1，0。

主函数程序(文件名：mlp.m)：

```
#include "stdio.h"
#include "math.h"
#include "mlp.c"
void main()
{ int i,j,k,l,l1,p,choice,layernum,iternum,sampnum;
  double miu,eps,e,ep;
  int n[10];
  char fname[14],chyn[1];
  static double w[500],b[100],y[100],d[100],delta[100];
  FILE *fp;
  void init(), input(), output(), update(), calcdelta();
  void feedforward(), readparameter(), saveparameter();
  double f(), fd() ;
  puts("1--Training,2--Output");
  scanf(" %d",&choice);
```

```
if ( choice == 1)
  { puts("Input name of training sample data file ");
    scanf("%s",fname);
    puts("Input the number of samples ");
    scanf("%d",&sampnum);
    puts("Input the learning rate: miu ");
    scanf("%lf",&miu);
    puts("Input overall error threshold: eps ");
    scanf("%lf",&eps);
    puts("Input the number of training (iteration) ");
    scanf("%d",&iternum);
    puts("Input layer number of MLP");
    scanf("%d",&layernum);
    puts("Input node number of each layer");
    for (l=0;l<layernum;l++)
      { printf("N(%d)=\n",l);
        scanf("%d",&n[l]);
      }
    k = n[0];
    for (l=1;l<layernum;l++)
      { if (k < n[l]) k = n[l]; }
    init(layernum,n,k,w,b) ;
    for (p=1;p<=iternum;p++)
      { if (p%100 == 0) printf("Iteration = %d\n",p);
        e = 0.0;
        if ((fp=fopen(fname,"r"))==NULL)
          { puts("\nCannot open the sample file");
            exit(0);
          }
        for (i=0;i<sampnum;i++)
          { for (j=0;j<n[0];j++)
              { fscanf(fp,"%lf",&y[j]); }
            for (j=0;j<n[layernum-1];j++)
              { fscanf(fp,"%lf",&d[j]); }
            feedforward(layernum,n,k,y,w,b);
            calcdelta(layernum,n,k,d,y,w,delta);
            update(layernum,n,k,y,delta,miu,w,b);
            ep = 0.0;
```

```
            for (l=0;l<n[layernum-1];l++)
              { l1 = layernum - 1;
            ep += (d[l]-y[(l1)*n[l1]-l])*(d[l]-y[(l1)*n[l1]-l]);
              }
            e = e + ep/2.0;
          }
        fclose(fp);
        if (e < eps)
          { printf("\nActual training number= %d\n",p);
            printf("\nThe overall error= %lf\n",e);
            break;
          }
      }
    printf("\nDo you want to save the network data (Y/N) ? ");
    scanf("%s",chyn);
    if (chyn[0] == 'Y' || chyn[0] == 'y')
      { saveparameter(layernum,n,k,w,b); }
  }
else
  { readparameter(&layernum,n,&k,w,b); }
do
  { input(n,y);
    feedforward(layernum,n,k,y,w,b);
    output(layernum,n,k,y) ;
    printf("\n\nDo you want to output again (Y/N) ? ");
    scanf("%s",chyn);
  } while (chyn[0] == 'Y'|| chyn[0] == 'y');
}
```

运行结果：

```
1——Training,2——Output    1
Input name of training sample data file    xor.dat
Input the number of samples    4
Input the learning rate: miu    0.5
Input overall error threshold: eps    0.0001
Input the number of training (iteration)    5000
Input layer number of MLP    3
Input node number of each layer
```

```
N(0)=2
N(1)=6
N(2)=1
Do you want to save the network data (Y/N) ?    y
Input name of network data file      mlp.dat
Input of MLP
Input pattern Y(0,0) = 0
Input pattern Y(0,1) = 0
Output of MLP
Y(2,0) = 0.018758
Do you want to output again (Y/N) ?   y
Input of MLP
Input pattern Y(0,0) = 0
Input pattern Y(0,1) = 1
Output of MLP
Y(2,0) = 0.978113
Do you want to output again (Y/N) ?   y
Input of MLP
Input pattern Y(0,0) = 1
Input pattern Y(0,1) = 0
Output of MLP
Y(2,0) = 0.979612
Do you want to output again (Y/N) ?   y
Input of MLP
Input pattern Y(0,0) = 1
Input pattern Y(0,1) = 1
Output of MLP
Y(2,0) = 0.023950
Do you want to output again (Y/N) ?   n
```

§1.2 离散 Hopfield 神经网络

一、功　能

离散 Hopfield 神经网络模型及其在联想记忆中的应用。

二、方法简介

离散 Hopfield 神经网络由 N 个神经元互连构成，每个神经元都与其他 $N-1$ 个神经

元相连。设第 i 个神经元的阈值为 θ_i，它接收来自第 j 个神经元的信号，其连接权值为 w_{ij}（假设 $w_{ij}=w_{ji}$，$w_{ii}=0$），于是神经元的状态演化方程为

$$v_i = f(\sum_{j=1}^{N} w_{ij}v_j - \theta_i) \quad , \quad i = 1,2,\cdots,N$$

其中 v_i 是第 i 个神经元的输出，$f(\cdot)$是硬限幅函数

$$f(x) = \begin{cases} 1 & , \quad x \geqslant 0 \\ -1 & , \quad x < 0 \end{cases}$$

上述离散 Hopfield 神经网络模型可用于联想记忆。所谓联想记忆，是指在学习过程中存入 M 个样本 $\{X^i\}(i=1,2,\cdots,M)$，然后，若输入 $\overline{X}=X^a+n$（其中 X^a 是 M 个学习样本之一，n 是噪声），则要求输出 $Y=X^a$。

通过合理选择离散 Hopfield 神经网络的权值，使网络的稳态恰好为联想记忆的一组状态，这样，当给网络输入一组相近的模式后，经过神经网络的动态演化，最终达到一个与初态在汉明距离意义上最近的状态，从而实现联想记忆。

用离散 Hopfield 神经网络实现联想记忆的具体步骤如下：

1. 根据给定的模式，利用外积规则，计算权值 W_{ij}（将 M 个模式存于网络中）

设 M 个模式为 $X^s=(x_1^s,x_2^s,\cdots,x_N^s)$，$s=1,2,\cdots,M$，其中 $x_i^s\in\{-1,+1\}$，它表示第 s 类模式的第 i 个元素，N 是网络节点数，M 是模式数。外积规则为

$$w_{ij} = \begin{cases} \sum_{s=1}^{M} x_i^s x_j^s & , \quad i \neq j \\ 0 & , \quad i = j \end{cases}$$

2. 用输入模式 $X=(x_1,x_2,\cdots,x_N)$ 作为网络的初始状态

$$u_i(0) = x_i \quad , \quad 1 \leqslant i \leqslant N$$

3. 离散 Hopfield 网络进行动态演化，即反复计算下式

$$v_i(t+1) = f(\sum_{j=1}^{N} w_{ij}v_j(t)) \quad , \quad i = 1,2,\cdots,N$$

直至达到稳态。此时，神经网络的状态描述了输入模式与已有模式的最佳匹配，从而实现了联想记忆。

4. 返回第 2 步，对下一个模式进行联想记忆。

三、使用说明

1. 子函数语句

```
void hnnd(w,sample,n,m)
```

2. 形参说明

w——双精度实型二维数组，体积为 n×n。w[i][j]表示第 j 个神经元的输出与第 i 个神经元的输入之间的连接权。

sample——双精度实型二维数组，体积为 m×n。sample[m][n]表示第 m 个模式的第 n 个元素。

n ——整型变量。神经元的数目。

m --整型变量。神经网络中联想记忆的模式数(在本节中,模式是0~9这10个数字,因此m=10)。

四、子函数程序(文件名:hnnd.c)

```
#include "stdio.h"
#include "conio.h"
#define  M         10
#define  N         512
#define  LN        32
#define  MAIN-W    window(1,20,80,25); textbackground(GREEN);
                   textcolor(WHITE);
#define  W1        window(5,1,5+LN-1,18); textbackground(WHITE);
                   textcolor(BLACK)
#define  W2        window(45,1,45+LN-1,18); textbackground(BLUE);
                   textcolor(YELLOW)
static last-x,last-y;
void hnnd(w,sample,n,m)
int m,n;
char (huge * w)[N], sample[][N];
{ int i,j,k,c;
  void recognize(char (huge * )[N], char [][N], int, int);
  for (i=0;i<n;i++)
    { for (j=0;j<i;j++)
        { w[i][j] = 0;
          for(k=0;k<m;k++)
            { w[i][j] += sample[k][j] * sample[k][i]; }
          w[j][i] = w[i][j];
        }
      w[i][i]=0;
    }
  while( 1 )
    { recognize(w,sample,n,m);
      wprintf("\n\nContinue to recognize another object? (Y/N)");
      while ((c=getch())!='N' && c!='n' && c!='Y' && c!='y');
      window(1,1,80,25); textbackground(BLACK); textcolor(WHITE);
      if (c=='N' || c=='n') break;
    }
}
```

```
static void recognize(w,sample,n,m)
int m,n;
char (huge *w)[N], sample[][N];
{ int flag,tmp,i,j,c;
  long k=0;
  double ratio;
  char t, object[N];
  int displyin(char [],int), displyout(char []);
  int inobject(char * ,int,FILE * );
  void correctdata(char,int,long);
  wprintf("\n\rInput data of object for recognition:\
    \nInput 0 ... %d (from sample) ",m-1);
  while (( ( c = getch() ) <'0' || c>m-1+'0') && c!='f');
  if (c>='0' && c<=m-1+'0')
    { wprintf("\nEnter the ratio of signal to noise (SNR):");
      cscanf(" %lf",&ratio);
      ratio /= ratio + 1;
      getch();
      for (i=0;i<n;i++)
        { t = (char)(rand()/32768.0+ratio);
          object[i] = ( ( sample[c-'0'][i] >0 ) ? 1 : 0)!t;
}
    }
  for (i=0;i<n;i++)
    { object[i] = (object[i] > 0) ? 1 : -1 ; }
  displayin(object,n);
  wprintf("\n\nPress any key to begin recognize ... ");
  getch();
  wprintf("\n");
  while( 1 )
    { flag = 0;
      for(i=0;i<n;i++)
        { tmp = 0;
          for(j=0;j<n;j++)
            { tmp += w[i][j] * object[j]; }
          tmp = ( tmp >= 0 ) ? 1 : -1;
          if ( object[i] != tmp )
```

```
                { object[i] = tmp;
                  flag = 1;
                  k++;
                  correctdata(object[i],i,k);
                }
            }
        if ( flag == 0 ) break;
      }
    displayout(object);
  }

static int displayout(object)
char object[];
{ int i;
  i = object[0];
  return(i);
}

static int displayin(object,n)
int n;
char object[];
{ int i;
  W1; gotoxy(1,2);
  for (i=0;i<n;i++)
     { putch( ( object[i] == -1 ) ? ' ' : '$' ); }
  W2; gotoxy(1,2);
  for (i=0;i<n;i++)
     { putch( ( object[i] == -1) ? ' ' : '$' ); }
}

static void correctdata(data,i,k)
char data;
int i;
long k;
{ W2; gotoxy(i%LN+1,i/LN+2);
  putch( ( data == -1 )? ' ' : '$' );
  wprintf("\n\r%5ld:object[%3d] is beening corrected;",k,i);
}
```

```
static int inobject(object,n,fp)
int n;
FILE *fp;
char object[];
{ unsigned i,j,tmp;
  for (j=0;j<n; )
    { if ( fscanf(fp,"%x",&tmp) == EOF ) return(EOF);
      for (i=0;i<8;i++)
        { object[j++]=((tmp & (0x80>>i)) > 0)? 1:-1;
          if (j == n) break;
        }
    }
  return(0);
}

static int wprintf(char *format,...)
{ int i,n,tmp;
  char buffer[256];
  va-list argptr;
  MAIN-W;
  gotoxy(last-x,last-y);
  va-start(argptr,format);
  n = vsprintf(buffer,format,argptr);
  va-end(argptr);
  for (i=0;i<256;i++)
    { switch(buffer[i])
        { case 0:
            tmp = 1;
            break;
          case '\n':
            putch('\r');
          default:
            putch(buffer[i]);
        }
      if (tmp == 1) break;
    }
  last-x = wherex();
```

```
    last-y = wherey();
    return(n);
  }
```

五、例 题

用离散 Hopfield 神经网络进行联想记忆，实现受噪声干扰数字的恢复。数字 0～9 由 16×32 点阵构成，并已存于数据文件 hnnd. dat 中。神经元数 $n=512$，模式数 $m=10$，信噪比 SNR=2。

主函数程序(文件名：hnnd. m)：

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#include "alloc.h"
#include "hnnd.c"
#define  M        10
#define  N        512
#define  LN       32
#define  MAIN-W   window(1,20,80,25); textbackground(GREEN);
                  textcolor(WHITE);
#define  W1       window(5,1,5+LN-1,18); textbackground(WHITE);
                  textcolor(BLACK)
#define  W2       window(45,1,45+LN-1,18); textbackground(BLUE);
                  textcolor(YELLOW)

main()
{ int i,j,k,n,m;
  static char sample[M][N], (huge *w)[N];
  FILE *fp;
  struct text-info text-data;
  int wprintf(char *,...);
  int inobject(char *,int,FILE *);
  void hnnd(char (huge *)[N],char [][N],int,int);
  gettextinfo(&text-data);
  if (text-data.currmode<2)
    { textmode(text-data.currmode+2); }
  if ((fp = fopen("hnnd.dat","r"))==NULL)
    { fprintf(stderr,"Can't open data file 'hnnd.dat'\n");
      exit(-1);
```

```
    }
  clrscr();
  W1; clrscr();
  W2; clrscr();
  MAIN-W; clrscr();
  wprintf("Input the number of nodes of neural networks( <= %d );\n",N);
  fscanf(fp,"%d",&n);
  wprintf("Input the number of training samples ( <= %d ) ;\n",M);
  fscanf(fp,"%d",&m);
  wprintf("The number of nodes and sample are %d and %d;\n",n,m);
  for (i=0;i<m;i++)
    { wprintf("\nInput the data of sample %d:\n",i+1);
      if (inobject(sample[i],n,fp) == EOF)
        { wprintf("Input data error or file is end. \n");
          exit(-1);
        }
      displayin(sample[i],n);
      wprintf("\nPress any key to continue ...");
      getch();
    }
  fclose(fp);
  randomize();
  if ((w=farcalloc(sizeof(*w),N))==NULL)
    { puts("no enough memory.");
      exit(-1);
    }
  hnnd(w,sample,n,m);
  window(1,20,80,25);
  clrscr();
  textbackground(WHITE);
  textcolor(BLACK);
}
```

运行结果:

受噪声干扰的数字"0"如图 5-1-1a 所示，联想记忆后的结果如图 5-1-1b 所示。受噪声干扰的数字"1"如图 5-1-2a 所示，联想记忆后的结果如图 5-1-2b 所示。

图5-1-1a 受噪声干扰的数字"0"

图5-1-1b 联想记忆后的数字"0"



图5-1-2a 受噪声干扰的数字"1"

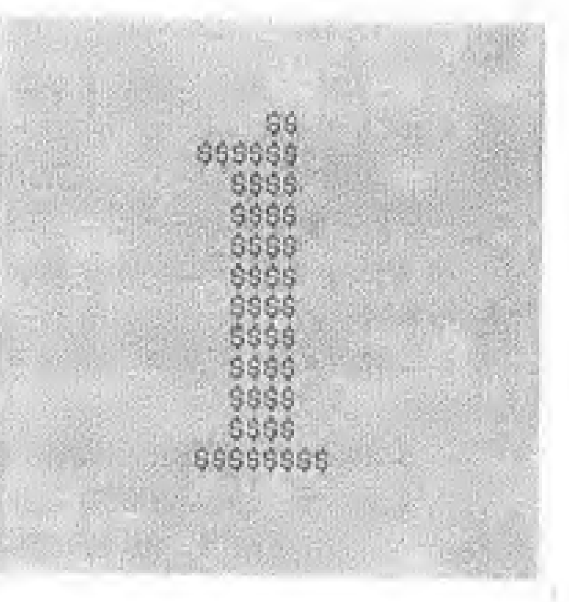


图5-1-2b 联想记忆后的数字"1"

§1.3　连续 Hopfield 神经网络

一、功　能

连续 Hopfield 神经网络模型及其在优化计算中的应用。

二、方法简介

设连续 Hopfield 神经网络共有 N 个神经元，第 i 个神经元的输入为 u_i，输出为 v_i，输入电阻为 R_i，输入电容为 C_i，阀值(偏置电流)为 I_i，第 j 个神经元的输出与第 i 个神经元的输入之间的连接权为 w_{ij}，则连续 Hopfield 神经网络可表示为

$$C_i\frac{du_i}{dt}=\sum_{j=1}^{N}w_{ij}v_j-\frac{u_i}{R_i'}+I_i$$

$$v_i=f(u_i)=\frac{1}{2}\left[1+\tanh\left(\frac{u_i}{u_0}\right)\right]$$

$$i=1,2,\cdots,N$$

其中 u_0 用于控制函数 $f(\cdot)$的形状，当 $u_0\to 0$ 时，$f(\cdot)$就成为硬限幅函数。R_i' 为

$$\frac{1}{R_i'}=\frac{1}{R_i}+\sum_{j=1}^{N}w_{ij}$$

在计算机上进行仿真计算时，上述微分方程组的求解常采用欧拉方法。

三、使用说明

1. 子函数语句

void hnnc(w,ii,n,u0,r,c,eps,h,v)

2. 形参说明

w——双精度实型二维数组，体积为 n×n。表示第 j 个神经元的输出与第 i 个神经元的输入之间的连接权。

ii——双精度实型一维数组，长度为 n。表示神经元的阀值(偏置电流)I_i。

n ——整型变量。神经元的数目。

u0——双精度实型变量。用于控制神经元函数 $f(\cdot)$的形状。当 $u_0\to 0$ 时，$f(\cdot)$成为硬限幅函数。

r——双精度实型变量。神经元的输入电阻。

c——双精度实型变量。神经元的输入电容。

eps——双精度实型变量。神经网络的计算精度。

h——双精度实型变量。欧拉方法的步长。

v——双精度实型一维数组，长度为 n。神经网络的输出。

四、子函数程序(文件名：hnnc.c)

```
#include "stdlib.h"
```

```
#include "math.h"
void hnnc(w,ii,n,u0,r,c,eps,h,v)
int n;
double h,c,r,u0,eps,w[],ii[],v[];
{ int i,j;
  double e, *u, *g, *v1, *s;
  u = malloc(n*sizeof(double));
  g = malloc(n*sizeof(double));
  v1 = malloc(n*sizeof(double));
  s = malloc(n*sizeof(double));
  h = h/c;
  for (i=0;i<n;i++)
    { g[i] = 1/r;
      for (j=0;j<n;j++)
        { g[i] = g[i] + w[i*n+j]; }
    }
  for (i=0;i<n;i++)
    { v1[i] = 0.0;
      u[i] = 0.0;
      v[i] = 0.5*(1+tanh(u[i]/u0));
    }
  do
    { for (i=0;i<n;i++)
        { s[i] = ii[i];
          for (j=0;j<n;j++)
            { s[i] = s[i] + w[i*n+j]*v[j]; }
        }
      for (i=0;i<n;i++)
        { u[i] = u[i] + h*(s[i] - g[i]*u[i]); }
      for (i=0;i<n;i++)
        { v[i] = 0.5*(1+tanh(u[i]/u0)); }
      for (i=0;i<n;i++)
        { e = fabs(v[i]-v1[i]);
          if ( e > eps )
            { for (j=0;j<n;j++)
                { v1[j] = v[j]; }
              break;
            }
```

```
        }
      } while ( e > eps );
    free(u);
    free(g);
    free(v1);
    free(s);
}
```

五、例　题

用连续 Hopfield 神经网络实现四位 A/D 变换器。神经网络的输入是模拟量 x，输出是由四个神经元给出的 4 位数字量，用 $\overline{V}_3\overline{V}_2\overline{V}_1\overline{V}_0$ 表示，这里 $\overline{V}_i \in \{0, 1\}$。经过推导，得到神经网络的连接权 w_{ij}和阀值 I_i 分别为

$$w_{ij} = -2^{(i+j)}$$

$$I_i = -2^{(2i-1)} + 2^i x$$

选取参数：神经元数 n=4，电阻 r=10，电容 c=1，u_0=0.002，欧拉方法的步长 $h=10^{-5}$，计算精度 eps=10^{-8}。神经网络从初始状态开始，经动态演化后达到稳定状态，从而得到 A/D 变换器的输出。

主函数程序(文件名:hnnc.m)：

```
#include "stdio.h"
#include "math.h"
#include "hnnc.c"
main()
{ int i,j,n
  double c,h,r,x,u0,eps;
  double w[16],ii[4],v[4];
  void hnnc();
  n = 4;
  r = 10.0;
  c = 1.0;
  u0 = 0.002;
  eps = 1.0e-8;
  h = 1.0e-5;
  while ( 1 )
    { printf("Please input X (-0.5~15.5): ");
      scanf("%lf",&x);
      if (x >= 16) exit(0);
      for (i=0;i<n;i++)
        { ii[i] = - pow(2.0,(2*i-1)) + pow(2.0,i) * x;
```

```
            for (j=0;j<n;j++)
              { if (i != j)
                  { w[i*n+j] = - pow(2.0,(i+j)); }
                else
                  { w[i*n+j] = 0.0; }
              }
          }
        hnnc(w,ii,n,u0,r,c,eps,h,v);
        for (i=n-1;i>=0;i--)
          { printf("        %6.1lf",v[i]); }
        printf("\n");
      }
  }
```

运行结果：

```
Please input X (0—15): 2
0.0         0.0         1.0         0.0
Please input X (0—15): 3.4
0.0         0.0         1.0         1.0
Please input X (0—15): 5.6
0.0         1.0         1.0         0.0
Please input X (0—15): 8.5
1.0         0.0         0.0         0.0
Please input X (0—15): 10.9
1.0         0.0         1.0         1.0
Please input X (0—15): 14.3
1.0         1.0         1.0         0.0
```

§1.4 Tank-Hopfield 线性规划神经网络

一、功　能

用于求解线性规划问题的 Tank-Hopfield 神经网络模型。

二、方法简介

线性规划的数学描述如下：

目标函数为

$$\min \sum_{i=1}^{N} a_i v_i$$

约束条件为

$$\begin{cases} d_{11}v_1 + d_{12}v_2 + \cdots + d_{1N}v_N \geqslant b_1 \\ d_{21}v_1 + d_{22}v_2 + \cdots + d_{2N}v_N \geqslant b_2 \\ \qquad\qquad \vdots \qquad\qquad\qquad \vdots \\ d_{M1}v_1 + d_{M2}v_2 + \cdots + d_{MN}v_N \geqslant b_M \end{cases}$$

其中 $v_i(i=1,2,\cdots,N)$ 是优化变量，a_i、d_{ji}、$b_j(i=1,2,\cdots,N;\ j=1,2,\cdots,M)$ 均是常数。

为求解上述线性规划问题，Tank 和 Hopfield 构造了神经网络模型，它由 N 个线性神经元 $g(\cdot)$ 和 M 个非线性神经元 $f(\cdot)$ 组成。设第 i 个线性神经元的输入为 v_i，输出为 u_i，输入电阻为 R_i，输入电容为 C_i，则 Tank-Hopfield 神经网络模型可描述为

$$C_i\frac{\mathrm{d}u_i}{\mathrm{d}t} = -a_i - \frac{u_i}{R_i'} - \sum_{j=1}^{M}d_{ji}f\left(\sum_{k=1}^{N}d_{jk}v_k - b_j\right)$$

$$v_i = g(u_i) = \beta u_i \quad ,\ \beta \text{ 为常数(电压放大倍数)}$$

$$i = 1,2,\cdots,N$$

其中 R_i' 为

$$\frac{1}{R_i'} = \frac{1}{R_i} - \sum_{j=1}^{M}d_{ji}$$

函数 $f(\cdot)$ 为

$$f(x) = \begin{cases} 0, & x \geqslant 0 \\ \alpha x, & x < 0 \end{cases}$$

其中 $\alpha>0$

在计算机上进行仿真计算时，上述微分方程组的求解常采用欧拉方法。

三、使用说明

1. 子函数语句

```
void hnnlp(a,d,b,m,n,r,c,eps,h,v)
```

2. 形参说明

a——双精度实型一维数组，长度为 n。线性规划目标函数的系数向量。

d——双精度实型二维数组，体积为 m×n。线性规划约束条件的左端系数矩阵。

b——双精度实型一维数组，长度为 m。线性规划约束条件的右端系数向量。

m ——整型变量。线性规划约束方程的个数，也是非线性神经元的数目。

n ——整型变量。线性规划优化变量的个数，也是线性神经元的数目。

r——双精度实型变量。线性神经元的输入电阻。

c——双精度实型变量。线性神经元的输入电容。

eps——双精度实型变量。神经网络的计算精度。

h——双精度实型变量。欧拉方法的步长。

v——双精度实型一维数组，长度为 n。神经网络的输出，即线性规划的解。

四、子函数程序(文件名:hnnlp.c)

```
#include "stdlib.h"
#include "math.h"
#include "uniform.c"
void hnnlp(a,d,b,m,n,r,c,eps,h,v)
int m,n;
double c,h,r,eps,a[],b[],d[],v[];
{ int i,j,k;
  long int seed;
  double e, *u, *v1, *s, *s1, *g;
  double uniform( double , double , long * );
  double gf(double), f(double);
  u = malloc(n * sizeof(double));
  g = malloc(n * sizeof(double));
  v1 = malloc(n * sizeof(double));
  s = malloc(n * sizeof(double));
  s1 = malloc(m * sizeof(double));
  h = h/c;
  for (i=0;i<n;i++)
    { g[i] = 1/r;
      for (j=0;j<m;j++)
        { g[i] = g[i] - d[j * n+i]; }
    }
  seed = 13579l;
  for (i=0;i<n;i++)
    { u[i] = uniform(0.0,1.0,&seed);
      v[i] = gf(u[i]);
      v1[i] = 0.0;
    }
  do
    { for (j=0;j<m;j++)
        { s1[j] = -b[j];
          for (k=0;k<n;k++)
            { s1[j] = s1[j] + d[j * n+k] * v[k]; }
        }
      for (i=0;i<n;i++)
        { s[i] = 0.0;
```

```
            for (j=0;j<m;j++)
              { s[i] = s[i] + d[j*n+i]*f(s1[j]); }
          }
        for (i=0;i<n;i++)
          { u[i] = u[i] - h*( a[i] + g[i]*u[i] + s[i] ); }
        for (i=0;i<n;i++)
          { v[i] = gf(u[i]); }
        for (i=0;i<n;i++)
          { e = fabs(v[i]-v1[i]);
            if ( e > eps )
              { for (j=0;j<n;j++)
                  { v1[j] = v[j]; }
                break;
              }
          }
      } while ( e > eps );
    free(u);
    free(g);
    free(v1);
    free(s);
    free(s1);
  }

  static double gf(x)
  double x;
  { double beta;
    beta = 100.0;
    return(x*beta);
  }

  static double f(x)
  double x;
  { double z,alpha ;
    alpha = 60.0;
    z = (x >= 0.0) ? 0.0 : alpha*x;
    return(z);
  }
```

五、例　题

用 Tank-Hopfield 神经网络求解下述线性规划问题：

目标函数为

$$c(v_1,v_2) = v_1 + v_2$$

约束条件为

$$\begin{aligned} v_1 &\geqslant -5 \\ -v_2 &\geqslant -5 \\ -5v_1 + 12v_2 &\geqslant -35 \\ -5v_1 - 2v_2 &\geqslant -35 \end{aligned}$$

选取参数：线性神经元数 n＝2，非线性神经元数 m＝4，电阻 r＝10，电容 c＝1，$\alpha=60$，$\beta=100$，欧拉方法的步长 $h=10^{-5}$，计算精度 $eps=10^{-4}$。神经网络从初始状态开始，经动态演化后达到稳定状态，从而得到线性规划的解。

主函数程序（文件名：hnnlp.m）：

```
#include "stdio.h"
#include "hnnlp.c"
main()
{ int i,m,n;
  double c,h,r,eps,v[4];
  static double a[2] = {1.0, 1.0};
  static double b[4] = {-5, -5, -35, -35};
  static double d[4][2] = {{1, 0},{0, -1},{-5, 12},{-5, -2}};
  void hnnlp();
  m = 4;
  n = 2;
  r = 10.0;
  c = 1.0;
  eps = 1.0e-4;
  h = 1.0e-5;
  hnnlp(a,d,b,m,n,r,c,eps,h,v);
  printf("The Solution of Linear Programming\n");
  for (i=0;i<n;i++)
    { printf("      v(%d) = %10.5lf\n",i,v[i]); }
}
```

运行结果：

线性规划的解为

```
v(0) =   -5.01807
v(1) =   -5.00764
```

参 考 文 献

[1] Law A M ,Kelton W D. Simulation Modeling and Analysis. New York:McGraw-Hill Book Company, 1982

[2] 朱华.系统模拟.北京:电子工业出版社,1990

[3] 程兴新,曹敏.统计计算方法.北京:北京大学出版社,1989

[4] Papoulis A. Probability, Random Variables, and Stochastic Processes. Auckland: McGraw-Hill Book Company, 1984

[5] Oppenheim A V, Schafer R W. Digital Signal Processing. Englewood Cliffs: Prentice-Hall Inc, 1975

[6] Rabiner L R, Gold B. Theory and Application of Digital Signal Processing. Englewood Cliffs: Prentice-Hall Inc, 1975

[7] Roberts R A, Mullis C T. Digital Signal Processing. Reading, MA: Addison-Wesley Publishing Company, 1987

[8] Proakis J G, Manolakis D G. Introduction to Digital Signal Processing. New York: Macmillan Publishing Company, 1988

[9] 中国电子学会、仪器仪表学会信号处理学会《数字信号处理程序库》编译组.数字信号处理程序库.北京:清华大学出版社,1983

[10] 北方交通大学信息科学研究所.近代数字信号处理通用程序.北京:科学出版社,1988

[11] Stearns S D, David R A. Signal Processing Algorithms. Englewood Cliffs: Prentice-Hall Inc, 1988

[12] 布赖姆 E O.快速富里叶变换.柳群译.上海:上海科学技术出版社,1979

[13] Burrus C S, Parks T W. DFT/FFT and Convolution Algorithms. New York: John Wiley & Sons Inc, 1985

[14] 蒋增荣等.快速算法.长沙:国防科技大学出版社,1993

[15] 张彦仲,沈乃汉.快速傅里叶变换及沃尔什变换.北京:航空工业出版社,1989

[16] Ahmed N, Rao K R. Orthogonal Transforms for Digital Signal Processing. Berlin: Springer-Verlag, 1975

[17] Elliott D F, Rao K R. Fast Transforms. New York: Academic Press Inc, 1982

[18] Blahut R E. Fast Algorithms for Digital Signal Processing. Reading, MA: Addison-Wesley Publishing Company, 1985

[19] Rao K R, Yip P. Discrete Cosine Transform. London: Academic Press Inc, 1990

[20] Duhamel P, Hollmann H. Split radix FFT algorithm. Electronics Letters, 1984(1)

[21] Sorensen H V, *et al.* On computing the split-radix FFT. IEEE Trans on Acoustics, Speech, and Signal Processing. 1986(1)

[22] Sorensen H V, *et al.* Real-valued fast Fourier transform algorithms. IEEE Trans on Acoustics,

Speech, and Signal Processing. 1987(6)

[23] Sorensen H V, *et al*. On computing the discrete Hartley transform. IEEE Trans on Acoustics, Speech, and Signal Processing. 1985(4)

[24] Narasimha M J, Peterson A M. On the computation of the discrete cosine transform. IEEE Trans on Communications, 1978(6)

[25] Lee B G. A new algorithm to compute the discrete cosine transform. IEEE Trans on Acoustics, Speech, and Signal Processing. 1984(6)

[26] Pei S C, Jaw S B. Computation of discrete Hilbert transform through fast Hartley transform. IEEE Trans on Circuits and Systems, 1989(9)

[27] 周耀华,汪凯仁.数字信号处理.上海:复旦大学出版社,1992

[28] 宗孔德,胡广书.数字信号处理.北京:清华大学出版社,1988

[29] Parks T W, Burrus C S. Digital Filter Design. New York: John Wiley & Sons Inc, 1987

[30] 何振亚.数字信号处理的理论与应用.北京:人民邮电出版社,1982

[31] 吴湘淇,聂涛.数字信号处理技术及应用.北京:中国铁道出版社,1986

[32] Kay S M. Modern Spectral Estimation. Englewood Cliffs: Prentice-Hall Inc, 1988

[33] Marple S L. Digital Spectral Analysis with Applications. Englewood Cliffs: Prentice-Hall Inc, 1987

[34] Proakis J G, *et al*. Advanced Digital Signal Processing. New York: Macmillan Publishing Company, 1992

[35] 王宏禹.随机数字信号处理.北京:科学出版社,1988

[36] 张贤达.现代信号处理.北京:清华大学出版社,1995

[37] 杨位钦,顾岚.时间序列分析与动态数据建模.北京:北京理工大学出版社,1988

[38] 徐士良.C常用算法程序集.北京:清华大学出版社,1994

[39] Martin W, Flandrin P. Wigner-Ville spectral analysis of nonstationary processes. IEEE Trans on Acoustics, Speech, and Signal Processing. 1985(6)

[40] Boashash B, Black P J. An efficient real-time implementation of the Wigner-Ville distribution. IEEE Trans on Acoustics, Speech, and Signal Processing, 1987(11)

[41] Rioul O, Vetterli M. Wavelets and signal processing. IEEE ASSP Magazine, 1991(4)

[42] Daubechies I. Orthonormal bases of compactly supported wavelets. Communications on Pure and Applied Mathematics, 1988, 41(7)

[43] Mallat S G. A theory for multiresolution signal decomposition: the wavelet representation. IEEE Trans on Pattern Analysis and Machine Intelligence, 1989(7)

[44] Mallat S G. Multifrequency channel decompositions of images and wavelet models. IEEE Trans on Acoustics, Speech, and Signal Processing, 1989(12)

[45] Cadzow J A. Foundations of Digital Signal Processing and Data Analysis. New York: Macmillan Publishing Company, 1987

[46] Haykin S. Adaptive Filter Theory. Englewood Cliffs: Prentice-Hall Inc, 1986

[47] Widrow B, Stearns S D. Adaptive Signal Processing. Englewood Cliffs: Prentice-Hall Inc, 1985

[48] 董士海等.图像格式编程指南.北京:清华大学出版社,1994

[49] Gonazalez R C, Wintz P. Digital Image Processing. Reading, MA: Addison-Wesley Publishing Company, 1977

[50] Rosenfeld A, Kak A C. Digital Picture Processing. New York: Academic Press Inc, 1976

[51] Jain A K. Fundamentals of Digital Image Processing. Englewood Cliffs: Prentice-Hall Inc, 1989

[52] 孙仲康，沈振康. 数字图像处理及其应用. 北京：国防工业出版社，1985

[53] 黄智. 图像处理和识别实用程序库. 天津：天津科学技术出版社，1987

[54] Otsu N. A threshold selection method from gray-level histograms. IEEE Trans on Systems, Man and Cybernetics, 1979(1)

[55] Huang T S, *et al*. A fast two-dimensional median filtering algorithm. IEEE Trans on Acoustics, Speech, and Signal Processing. 1979(1)

[56] Huang T S (Ed.). Two-Dimensional Digital Signal Processing Ⅱ: Transforms and Median Filters. Berlin: Springer-Verlag, 1981

[57] Frei W, Chen C C. Fast boundary detection: a generalization and a new algorithm. IEEE Trans on Computers, 1977(10)

[58] Pavlidis T. Algorithms for Graphics and Image Processing. Rockville: Computer Science Press, 1982

[59] Stefanelli R, Rosenfeld A. Some parallel thinning algorithms for digital pictures. Journal of ACM, 1971(2)

[60] 马建波. C语言图像处理程序集. 北京：海洋出版社，1992

[61] Zurada J M. Introduction to Artificial Neural Systems. New York: West Publishing Company, 1992

[62] Lippmann R P. An introduction to computing with neural nets. IEEE ASSP Magazine, 1987(1)

[63] 杨行峻，郑君里. 人工神经网络. 北京：高等教育出版社，1992

[64] 张立明. 人工神经网络的模型及其应用. 上海：复旦大学出版社，1993

[65] Tank D W, Hopfield J J. Simple "neural" optimization networks: an A/D converter, signal decision circuit, and a linear programming circuit. IEEE Trans on Circuits and Systems, 1986(5)

[66] Kennedy M P, Chua L O. Unifying the Tank and Hopfield linear programming circuit and the canonical nonlinear programming circuit of Chua and Lin. IEEE Trans on Circuits and Systems, 1987(2)

[67] Kennedy M P, Chua L O. Neural networks for nonlinear programming. IEEE Trans on Circuits and Systems, 1988(5)